Spock Framework

Testes automatizados ágeis para Java e REST

Edição atualizada

Casa do Código | alura

JOSÉ YOSHIRIRO

Sumário

- ISBN
- Agradecimentos
- Sobre o livro
- Prefácio
- Comentários sobre o livro
- 1. Por que usar Spock framework?
- 2. Preparando o ambiente para trabalhar com Spock
- 3. Primeiros testes automatizados com Spock
- 4. Anatomia de um teste Spock
- 5. Data-Driven Testing – Facilitando os testes de múltiplos cenários
- 6. Exception Conditions - Testes à espera de exceções
- 7. Testando com a ajuda de mocks
- 8. Testando RESTful APIs
- 9. Recursos avançados
- 10. Relatórios profissionais com Spock Reports Extension
- 11. Testando metaprogramação (anotações)
- 12. Testes de integração em projetos Spring
- 13. Apêndice A - Guia de Groovy para desenvolvedores Java
- 14. Apêndice B - Perguntas frequentes sobre o Spock
- 15. Referências

ISBN

Impresso: 978-85-5519-373-6

Digital: 978-85-5519-374-3

Caso você deseje submeter alguma errata ou sugestão, acesse http://erratas.casadocodigo.com.br.

Dados Internacionais de Catalogação na Publicação (CIP)
(Câmara Brasileira do Livro, SP, Brasil)

Yoshiriro, José
Spock framework : testes automatizados ágeis para Java e REST / José Yoshiriro. -- São Paulo : AOVS Sistemas de Informática, 2024.

ISBN 978-85-5519-373-6

1. Java (Linguagem de programação) 2. Programação (Computadores) 3. Software - Desenvolvimento - Testes 4. Software - Manuais I. Título.

24-199337 CDD-005.1068

Índices para catálogo sistemático:

1. Software : Desenvolvimento : Projetos : Gerenciamento : Ciência da computação 005.1068

Eliane de Freitas Leite - Bibliotecária - CRB 8/8415

Código-fonte dos projetos

Todos os códigos desta obra estão em projetos no repositório https://github.com/jyoshiriro/exemplos-livro-spock-framework/.

Agradecimentos

Agradeço primeiramente a Deus, pelo dom da vida, por todas as oportunidades que permitiu que eu tivesse e por todos os livramentos que sempre me proporcionou.

Agradeço a minha mãe, dona Mizuko, que sempre fez de tudo para que eu tivesse uma educação de qualidade ao alcance de suas possibilidades financeiras. Sua insistência no meu foco nos estudos foi determinante para que eu me tornasse o profissional que sou. Agradeço a meu pai, José (*In Memoriam*), que deu todo o apoio para que eu somente estudasse durante a infância e adolescência.

Agradeço a minha amada esposa e companheira de muitos anos, Danusa. Seu jeito carinhoso e meigo são uma contínua inspiração para meu cotidiano pessoal e profissional. Ela me deu um filho maravilhoso, o Tadashi, a quem amo sem medida.

Agradeço ao professor Manoel José dos Santos Sena, que foi o primeiro a me sugerir que escrevesse um livro, em 2008. Agradeço ao professor Eduardo Martins Guerra, que foi o primeiro a publicar meus textos, em forma de artigos para a antiga revista MundoJava, tornando-me conhecido nacionalmente.

Agradeço ao meu ex-aluno Ariel M. Lira, que me ajudou na melhoria das imagens para a 2ª edição, de agosto de 2019.

E agradeço a você que adquiriu esta obra. Muito obrigado pela confiança. Espero que este trabalho ajude você a criar testes fantásticos!

Sobre o livro

Esta obra destina-se a estudantes e profissionais de desenvolvimento de software, principalmente para a plataforma Java. Profissionais que trabalham exclusivamente criando testes também são público para este livro, pois o Spock também permite criar testes para REST APIs.

Para melhor acompanhar o conteúdo desta obra, são necessárias noções intermediárias de programação com a plataforma Java, tais como: configuração do ambiente, orientação a objetos, coleções e tratamento de exceções.

Esta obra contém capítulos sobre testes para REST APIs, Spring Boot e Spring MVC. Esses capítulos são destinados a profissionais com o mínimo de conhecimento sobre essas tecnologias.

Conceitos básicos sobre testes de software não são necessários, pois são revisados em vários tópicos do livro.

Caso você, leitor ou leitora, tenha pouco ou nenhum conhecimento sobre a linguagem Groovy, pode recorrer ao *Apêndice A - Guia de Groovy para desenvolvedores Java*.

A versão do Spock abordada neste livro foi a **2.3-groovy-4.0**, lançada em 29 de setembro de 2022, sendo a mais recente versão estável no momento da escrita deste livro.

Sobre o autor

José Yoshiriro Ajisaka Ramos, mais conhecido como **Yoshi**, nasceu em julho de 1981 em Belém-PA. É bacharel em Sistemas de Informação e especialista em Engenharia de Sistemas. Possui

certificações internacionais PMI (ACP), OMG (OCUP Fundamental) e ORACLE (SCBCD 5, SCWCD 5, SCJP 6 e SOA Suite 11g Certified Implementation Specialist).

Atua profissionalmente com desenvolvimento de software desde 2002, mas programa desde 1994. Atuou na criação e/ou manutenção de sistemas usados em organizações como TJE-PA, BrasilPrev, Rico, Caixa Econômica Federal, DETRAN-PA, BASA, Citibank, American Red Cross (EUA) e OneBlood (EUA). Também colaborou com os projetos *open source* Struts 2 e Grails. Programar é uma de suas grandes paixões.

É professor desde 2003, atuando como instrutor em vários cursos de programação. Como docente em nível superior, já trabalhou para UAB/IFPA, Faculdade Ipiranga e FIAP. No início de 2014, passou a compor o corpo docente da faculdade São Paulo Tech School - SPTech. Certamente já teve mais de mil alunos. Publicou vários artigos sobre programação nas revistas *MundoJava* e *GroovyMag* (EUA). É autor de outro livro, o *Deixe seu código limpo e brilhante - Desmistificando Clean Code com Java e Python*, também publicado pela Casa do Código. Palestrou em diversos eventos de tecnologia, como na Virada Tecnológica de São Paulo 2017, nos TDC São Paulo de 2014, 2015, 2017 e 2019, e no FISL de 2015. Ensinar é sua grande paixão.

LinkedIn: https://linkedin.com/in/jyoshiriro

Prefácio

Por Prof. Dr. Eduardo Martins Guerra*

Há cerca de 20 anos, começava um movimento no mundo da engenharia de software que contestava a forma como as coisas estavam sendo feitas e propunha novos valores. Surgiam o que chamamos hoje de métodos ágeis!

Todos os princípios e técnicas do desenvolvimento ágil são baseados em uma premissa: o código existente deve ser fácil de ser modificado. Porém, isso não vem de graça! É necessária muita disciplina nas práticas utilizadas para que isso realmente seja uma realidade. Além de ser preciso cuidado constante para manter o código limpo e desacoplado, os testes automatizados também possuem um importante papel nessa questão.

A presença de testes automatizados é importante para dar segurança em qualquer modificação do código. Seja para refatorá-lo, buscando frequentemente mantê-lo desacoplado, simples e legível, seja para inserir novas funcionalidades, os testes permitem uma detecção mais rápida quando algum comportamento anterior é quebrado. São esses testes que ajudam a manter baixo o custo de mudança, o que é fundamental, por exemplo, para viabilizar iterações curtas e com entregas frequentes.

Porém, a princípio muita gente não se dá conta de que testar não é nada fácil! Principalmente quando estamos falando de testes automatizados. Um software interage com diversos recursos dentro de seu ecossistema e provê diferentes tipos de interface para a execução de suas funcionalidades. São questões técnicas como: mapeamento de classes para o banco de dados, acesso a serviços por REST APIs ou mesmo dependências de sistemas externos que precisam ser simulados.

Ter que lidar com essas questões nos testes faz muita gente travar! Para superar essas dificuldades, é importante conhecer uma

ferramenta adequada juntamente com as técnicas de testes. É por isso que este livro que você tem em mãos a respeito do Spock framework contém um conhecimento muito valioso! Além de ensinar a utilizar as funcionalidades desse framework, ele ainda apresenta as técnicas de teste, juntamente com exemplos didáticos e focados em problemas do mundo real para tecnologias que são padrão de mercado.

Conheci o Yoshi quando eu era editor-chefe da revista *MundoJava*, época em que ele sempre entrava em contato para propor artigos com temas atuais e extremamente relevantes para o mercado. Seus artigos eram sempre muito apreciados pela equipe técnica da revista e pelos leitores devido à didática e à clareza com as quais apresentava os temas. Neste livro, ele acertou em cheio em uma temática na qual muita gente no mercado ainda precisa amadurecer. Tenho certeza de que a aplicação dos conceitos deste livro pode ser decisiva para o sucesso de muitos projetos! *Bora testar, galera!*

* **Eduardo Martins Guerra** é pesquisador do INPE. Foi professor no ITA, onde concluiu sua graduação, mestrado e doutorado. Suas pesquisas focam no design, arquitetura e teste de software. Foi editor-chefe da revista *MundoJava* por vários anos. É autor/coautor de vários livros na área de computação.

Comentários sobre o livro

"Escrito de forma bem clara e didática, este livro não se restringe a apenas ensinar como utilizar o framework Spock, mas também apresenta de forma bem abrangente os principais conceitos relacionados a testes de software, bem como um comparativo entre as ferramentas de testes existentes no mercado. O uso do Spock é exemplificado em diversas plataformas e frameworks, incluindo Spring e REST APIs. Leitura imprescindível para todas as pessoas que prezam pela qualidade de seu software."

Célia Taniwaki. Professora na faculdade SPTech. Mestra em Engenharia Elétrica pela USP e especialista em Engenharia Elétrica pela Mei University do Japão. Foi analista/desenvolvedora na Itautec por 13 anos. Já traduziu vários livros e revistas de computação.

"Este livro do Yoshi é um ensaio tecnológico sobre testes de software. De forma equilibrada, explora conceitos importantes sobre testes de softwares e de várias ferramentas de testes relevantes no mercado. Toda pessoa desenvolvedora deve conhecer este livro, pois é rico de exemplos e explica, detalhadamente, como configurar e aplicar as ferramentas nas principais IDEs e linguagens de programação da plataforma Java."

Daniel Couto Gatti. Diretor da Faculdade de Ciências Exatas e Tecnologia da PUC-SP. Doutor em Educação Matemática e mestre em Comunicação e Semiótica pela PUC-SP. É autor de vários livros na área de computação.

"Este livro que você tem em mãos a respeito do Spock framework contém um conhecimento muito valioso! Além de ensinar a utilizar as funcionalidades desse framework, ele ainda apresenta as técnicas de teste, juntamente com exemplos didáticos e focados em problemas do mundo real para tecnologias que são padrão de mercado. Neste livro, o Yoshi acertou em cheio em uma temática na

qual muita gente no mercado ainda precisa amadurecer. Tenho certeza de que a aplicação dos conceitos deste livro pode ser decisiva para o sucesso de muitos projetos!"

Eduardo Martins Guerra. Pesquisador na Free University of Bolzen-Bolzano. Doutor e Mestre em Engenharia Eletrônica e Computação pelo ITA, onde também foi professor. Foi editor-chefe e articulista da revista *MundoJava*. É autor de vários livros na área de computação.

"Fazia muito tempo que esperava por um livro como este do Yoshiriro: é relevante, enriquece o leitor e vai além de um mero 'tutorial de Spock'. Aprendi coisas novas durante a sua leitura, coisas que vão além da documentação oficial.

Se você quer entender como o Spock funciona — um framework que vejo muita gente aprender por aí dando murro em ponta de faca —, creio que o livro é este. É bem fundamentado, bem escrito, didático e direto ao ponto. Você aprenderá não só como o Spock funciona, mas também algumas coisas a mais que valerão a pena no final. É o manual que faltava para o framework: que bom que foi escrito aqui no Brasil e em português para que torne essa poderosa ferramenta mais acessível."

Henrique Lobo Weissmann (Kico). Sócio-fundador da itexto, empresa de desenvolvimento de sistemas. Referência nacional em Spring e Groovy/Grails, é autor de livros sobre essas tecnologias e criador da comunidade Grails Brasil.

"Tive o privilégio de acompanhar parte do processo de formação do José Yoshiriro, como professor na graduação. Ele sempre foi muito entusiasmado com a tecnologia. Mas, além disso, tinha desde cedo muito forte o sentido da organização e da estruturação de ideias. Desta forma, é capaz de expressar conceitos Spring e complexos de maneira que as pessoas possam compreender facilmente. Ao ler este livro e conversar com o Yoshiriro, fico muito feliz em ver o quanto ele foi capaz de progredir ainda mais nestas habilidades com

o passar do tempo. Destaco a forma como foram escolhidos e colocados para o(a) leitor(a) os diversos tipos de teste, que formam a linha de base das aplicações do livro. Com certeza, esta obra é uma leitura que, além de levar ao conhecimento técnico específico, é inspiradora no sentido de formas inovadoras de comunicação."

Manoel José dos Santos Sena. Diretor Executivo do Mundo Digital Interativo - MDI e professor na faculdade FACI Wyden. Mestre em Concepção Mecânica e doutor em Engenharia Mecânica pela Université Joseph Fourier - Grenoble I, UJF da França.

"O livro está muito bem escrito e gostei muito da forma como o conhecimento é construído: saindo do básico, ensinando como configurar o ambiente, até chegar a testes com RESTful APIs. Mesmo sem conhecer Groovy, fui capaz de entender o funcionamento dos testes. O Apêndice A está muito bom e foi o suficiente para um desenvolvedor .NET entender o básico de Groovy."

Paul Marques. Programador desde 2015. Atualmente, é desenvolvedor na Ouribank, onde trabalha com web services, sistemas web e desktop e automação de sistemas. É tecnólogo em Análise e Desenvolvimento de Sistemas pela faculdade SPTech.

"De maneira muito intuitiva, apresenta uma disciplina fundamental para toda pessoa desenvolvedora de software. Esclarece aos mais apressados e dá as ferramentas para quem quer mais profundidade. Foca nos benefícios de usar a ferramenta de acordo com o problema e torna praticamente transparente a apresentação do uso de uma linguagem de programação não tão comum no Brasil, mas de adoção extremamente simples em qualquer aplicação Java. Percebe o tamanho do desafio cumprido na proposta deste livro?! Fico muito contente em poder contar com esse nível de material em português e ver a didática do Yoshiriro agora materializada além dos limites da sala de aula e projetos em que contribui. E que venham os próximos!"

Victor Franzonatto. Engenheiro de software sênior no Mercado Livre. Trabalha com desenvolvimento de software desde 2006. Já liderou projetos em empresas como IBM, Porto Seguro e BrasilPrev. Já publicou e traduziu vários artigos sobre tecnologia.

Capítulo 1
Por que usar Spock framework?

O Spock framework é uma ferramenta muito poderosa, que pode ser a grande aliada de sua equipe na construção mais rápida e simples de testes melhores e menos verbosos. Ela é uma ferramenta completa que dispensa a configuração de bibliotecas adicionais para implementação da maioria dos tipos de testes automatizados. Entre as funcionalidades que tornam o Spock tão completo, algumas que ganharam destaque nesta obra são: criação de simples testes unitários; uso de *Mocks*; o sensacional recurso de *Data-Driven Testing*; testes de integração para Spring e testes funcionais para REST APIs.

Neste primeiro capítulo, veremos os principais motivos que tornam relevante a adoção do Spock framework na criação de testes automatizados. A própria importância da criação desses testes será revisada e reforçada. Um breve histórico do Spock e informações sobre a adoção serão abordados para demonstrar quão consolidada essa ferramenta está.

1.1 Por que criar testes automatizados?

Conheça os principais motivos que demonstram a importância da criação de **testes automatizados** em um projeto de software.

Porque qualidade sem testes é pura sorte

O termo **"teste"** denota **qualidade** em qualquer área de conhecimento humano. Governos, por exemplo, só liberam a

comercialização de medicamentos que foram devidamente *testados*. Você usaria um medicamento sabendo que não foi devidamente testado? Novos modelos de carros só começam a ser fabricados e vendidos após vários *testes*. Você dirigiria um carro sabendo que não foi suficientemente testado? Não poderia ser diferente com software: para a criação de um programa de qualidade, é preciso submetê-lo a quantos *testes* forem necessários.

Porque testes manuais são importantes, mas os automatizados são indispensáveis

Testes manuais, ou seja, **executados integralmente por usuários humanos**, possuem sua importância. Avaliações sobre usabilidade, aparência da interface, disposição e tamanho de elementos gráficos, dentre outros de natureza subjetiva, ainda são feitas melhor por pessoas.

Supondo situações de pouco tempo, pouco orçamento e pouco conhecimento técnico da parte da equipe de desenvolvimento, os testes manuais seriam a melhor, senão única, alternativa. Quando não se tem acesso ao código do sistema, dependendo da sua natureza, a única forma de testar é manualmente (é o caso de alguns aplicativos nativos para Windows, por exemplo).

Testes automatizados são **pequenos programas criados por pessoas programadoras para testar outros programas**. Uma vez que não é um humano quem os executa, alguns benefícios imediatos são:

- **Velocidade infinitamente superior** na execução dos testes;
- **Chances de erro quase nulas** no resultado do teste;
- Como são programas, ou seja, possuem um código-fonte, servem como **evidência de que foram feitos testes**;
- As **evidências dos resultados** dos testes são geradas automaticamente;
- Podem **simular situações impossíveis para humanos**, como milhões de execuções simultâneas, ou testes com valores

aleatórios, por exemplo;
- **Todos os testes programados serão executados**. Assim, não ocorre a omissão ou execução incorreta de alguns testes por cansaço, ansiedade, preguiça, má-fé etc.;
- Podem ser **reexecutados sempre que necessário**. Isso é muito útil para saber se mudanças em uma parte código-fonte do projeto não tiveram algum efeito colateral inesperado em outra parte.

Outro benefício interessante dos testes automatizados é que, durante o processo de criação deles, pode-se encontrar problemas de *design* no código testado, tais como excesso de métodos estáticos, alto acoplamento, baixa coesão etc. Um código difícil de ser testado é, provavelmente, um código que precisa ser refatorado.

1.2 Testes que podemos criar com Spock

O Spock framework é uma ferramenta que permite a criação de alguns tipos de **testes automatizados**, que são:

Testes unitários (ou de unidade)

Testes isolados para pequenas partes de um programa. Normalmente, cria-se uma classe de teste unitário para cada classe do projeto. Fazendo uma analogia com a fabricação de um carro, seria como testar, isoladamente, cada pequena peça que compõe o veículo.

Testes de integração

Usado para testar a integração entre duas ou mais partes do programa ou a integração de uma parte do programa a um elemento externo (banco de dados, REST API etc.). É um complemento aos testes unitários, pois uma parte que funciona perfeitamente sozinha não terá, necessariamente, um funcionamento perfeito em conjunto

com outra(s). Na analogia com a fabricação de um carro, seria como testar a integração entre todas as peças que compõem o câmbio de um carro, ou seja, testar o câmbio montado.

Testes funcionais (ou de sistema)

Teste que valida o comportamento final, ou seja, se as saídas do sistema estão funcionando conforme o esperado. Por exemplo:

- Se o sistema em questão for um *website*, um teste poderia ser sobre uma de suas páginas HTML;
- Se o sistema for uma REST API, um teste poderia ser sobre um de seus *endpoints*.

Fazendo uma analogia com a fabricação de um carro, seria como testar um exemplar montado em situações como: Ele se comporta nas curvas como esperado? Seu consumo de combustível na cidade é o esperado? O tempo e distância levados para parar estando a 40 km/h são os esperados?

No caso dos **testes unitários** e **de integração**, o Spock permite criar testes para projetos que usam qualquer linguagem da **plataforma Java**. Quanto a **testes funcionais**, é possível criar testes para **REST APIs** e para **websites**.

No livro *Testes automatizados de software: Um guia prático*, escrito por Maurício Aniche e publicado pela Casa do Código, os conceitos e a importância dos testes automatizados de software são mais profundamente detalhados.

1.3 Por que usar Spock para projetos Java se temos JUnit, TestNG, Mockito etc.?

Existem várias bibliotecas de teste para Java, como a **JUnit** (considerada a padrão de mercado) e a **TestNG**. Elas permitem criar

os mesmos tipos de testes que o Spock. Por que então usar mais uma? Veja a seguir os 7 principais motivos para adotar essa ferramenta.

1. Spock não é só mais uma ferramenta – é uma por todas

O primeiro motivo é que o Spock é um framework **completo** para a escrita de testes para o qual se propõe. Ele funciona *out of the box*, ou seja, basta configurá-lo em seu projeto e você já poderá criar praticamente todo tipo de teste unitário, de integração ou funcional, não precisando de mais dúzias de outras bibliotecas de testes. Até a criação de *Mocks* e *Stubs* pode ser feita sem a inclusão de bibliotecas como **Mockito** e **PowerMock**. Uma interessante consequência disso é que basta ter em mãos um livro e/ou ler apenas uma documentação oficial.

Testes que falham indicam melhor por que falham

Os testes que falham em Spock possuem uma descrição bem mais completa e assertiva do motivo da falha, evitando que se executem as classes de testes em *modo Debug* para, por exemplo, achar o que ocasionou um `NullPointerException` ou com quantos elementos uma lista chegou ao final do teste. No capítulo 3, *Primeiros testes automatizados com Spock*, é demonstrado esse comportamento.

Data-Driven Tests

Existem situações em que é necessário testar várias combinações de entradas para o mesmo cenário. Por exemplo: para testar se um método que deve validar os valores de um contracheque para indicar se ele é valido ou se há fraude. Em um contracheque há vários valores de recebimentos e descontos. Imagine testar as várias combinações possíveis que podem resultar em válido ou inválido. O Spock possui um recurso chamado ***Data-Driven Tests***, que facilita muito a criação de testes para situações como esta. Esse recurso é abordado no capítulo 5, *Data-Driven Testing – Facilitando os testes de múltiplos cenários*.

2. Groovy em vez de Java: mais testes com menos código

Spock usa como linguagem de programação a Groovy, que é bem menos verbosa que Java, permitindo que muitas operações comuns do cotidiano como iterações, criação de *Mocks*, manipulação de coleções, acesso a métodos privados (logo, possibilidade de testá-los) e conversões de tipos sejam feitas de forma bem menos burocrática (veja o *Apêndice A – Guia de Groovy para desenvolvedores Java*).

3. Maven, Gradle e plugins das IDEs completas são 100% compatíveis

A execução de testes automatizados escritos em Spock é realizada automaticamente no processo de `build` do **Maven** ou **Gradle**, e com a funcionalidade de executar um ou mais arquivos de testes, as IDEs **IntelliJ** e **Eclipse** executam normalmente os testes criados com Spock. Isso porque os testes são interpretados por essas ferramentas como testes escritos em **JUnit**.

4. Testes que falham indicam melhor por que falham

Os testes que falham em Spock possuem uma descrição bem mais completa e assertiva do motivo da falha, evitando que se executem as classes de testes em *modo Debug* para, por exemplo, achar o que ocasionou um `NullPointerException` ou com quantos elementos uma lista chegou ao final do teste. No capítulo 3, *Primeiros testes automatizados com Spock*, é demonstrado esse comportamento.

5. Data-Driven Tests

Existem situações em que é necessário testar várias combinações de entradas para o mesmo cenário — por exemplo, para testar um método que deve validar os valores de um contracheque para indicar se ele é válido ou se há fraude. Em um contracheque, há vários valores de recebimentos e descontos. Imagine testar as várias combinações possíveis que podem resultar em válido ou inválido. O

Spock possui um recurso chamado ***Data-Driven Tests***, que facilita muito a criação de testes para situações como essa. Esse recurso é abordado no capítulo 5, *Data-Driven Testing – Facilitando os testes de múltiplos cenários*.

6. Adoção

Segundo dados do portal Maven Repository (https://mvnrepository.com), no momento da escrita desta obra, o `spock-core` é usado por 2.856 artefatos e está na posição 17 na categoria *Testing Frameworks & Tools*. Esses números podem ser vistos na figura a seguir e consultados de forma atualizada em https://mvnrepository.com/artifact/org.spockframework/spock-core/:

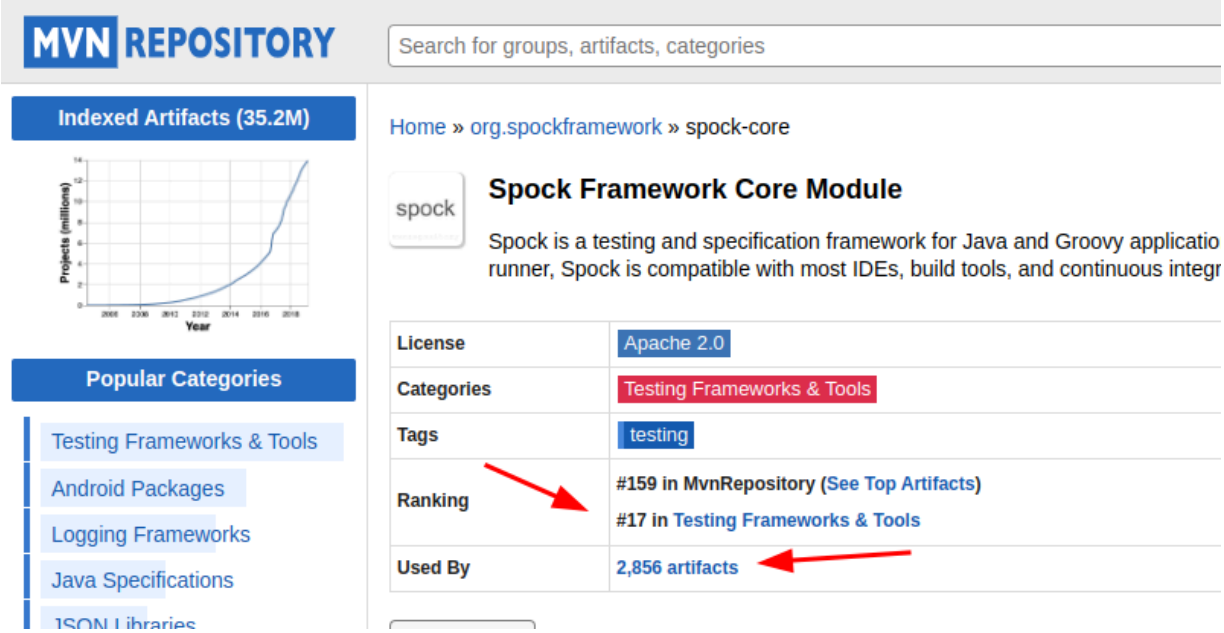


Figura 1.1: Repositórios com Spock framework em suas dependências. Fonte: Maven Repository

Um fato interessante ocorreu no final de 2013: foi lançada a versão 2.3 do framework **Grails**, um dos mais populares da plataforma Java. A partir dessa versão, o Spock passou a ser sua ferramenta padrão para a criação de testes unitários e de integração. Veja mais em https://grails.github.io/grails2-doc/2.3.0/guide/introduction.html#whatsNew23.

7. Constante evolução

O Spock framework foi criado por Peter Niederwieser e Luke Daley em 2008, mas seu primeiro *release* foi em 2009. Teve alguns hiatos sem novas versões em 2011, 2013 e 2014, mas desde 2015 tem havido atualizações com uma boa frequência. Seu histórico completo de *releases* pode ser encontrado em https://mvnrepository.com/artifact/org.spockframework/spock-core.

Desde seu lançamento em 2009 até dezembro de 2022, já foram **44 *releases***, sendo **7 apenas em 2022**. Em agosto de 2023, foram quase 30 *commits* para sua branch *master*. É possível constatar o quão ativo é o projeto vendo a atividade em torno do seu código-fonte no GitHub (https://github.com/spockframework/spock).

Conclusão

Neste capítulo, vimos os principais motivos que justificam a adoção do framework Spock, assim como um breve histórico e apanhado sobre seu estado atual de adoção. No próximo capítulo, veremos como **configurar um projeto** para o uso do Spock.

Capítulo 2
Preparando o ambiente para trabalhar com Spock

A configuração de um ambiente Spock consiste basicamente em preparar o ambiente para a linguagem Groovy e incluir algumas dependências no projeto onde o Spock será usado. Neste capítulo, você verá como realizar essas duas ações.

2.1 Configurando o Groovy nas principais IDEs Java

Como o Spock é uma ferramenta para a escrita de testes em projetos para Java, é natural que sejam usadas as principais IDEs dessa plataforma. Não é necessária a instalação de plugin específico para Spock, mas devemos habilitar a capacidade de editar e compilar arquivos Groovy na IDE. **IntelliJ**, **Eclipse** e **NetBeans** possuem suporte para Groovy, conforme detalhado a seguir.

Eclipse

Existe um conjunto de plugins para Groovy, o **Groovy-Eclipse**, que, historicamente, possui versão compatível com a última ou penúltima versão do Eclipse. Para instalá-lo, o processo é o mesmo de instalação de qualquer plugin: basta adicionar o *Update Site* conforme a versão do Eclipse desejada. As versões disponíveis do Groovy-Eclipse constam na sua *wiki* (https://github.com/groovy/groovy-eclipse/wiki).

Dentre os plugins que serão listados para instalação, os necessários para o uso do Spock são:

- Eclipse Groovy Development Tools;
- Groovy Compiler 4.0;
- Groovy-Eclipse M2E integration.

IntelliJ IDEA

O **IntelliJ IDEA**, mesmo na versão *Community Edition*, já conta com plugin para Groovy embarcado. Assim, não é necessária a instalação de qualquer complemento nessa IDE. Esse plugin, como o do Eclipse, está em constante atualização.

Se você optar por usar essa IDE, talvez não consiga executar individualmente os testes escritos em Spock caso abra um projeto apenas com o `pom.xml` e a pasta `src`. Caso isso ocorra, apenas execute o `build` do projeto (menu `Build` > `Build Project`).

NetBeans

O NetBeans, desde sua versão 9, já vem com plugin **Groovy**, criado e mantido pela própria equipe do NetBeans.

2.2 Uso de Groovy em editores de código simples (VS Code, Sublime Text, Vim etc.)

Caso você prefira usar uma IDE "leve", ou seja, editores de código como **VS Code**, **Sublime Text**, **Vim** e similares, é necessário:

1. Instalar os plugins para Java e Groovy de sua preferência;
2. Instalar o Groovy e seu *Development Kit* (GDK) no sistema operacional.

A versão do Groovy usada nos códigos deste livro foi a **4.0.x**. A seguir, veremos como instalá-los com as ferramentas **SDKMAN!** e **asdf**.

Groovy no Linux e macOS – SDKMAN!

Para Linux e macOS, uma das ferramentas recomendadas é a **SDKMAN!** (http://sdkman.io/), que é uma *Command Line Interface* (CLI) que permite a instalação do Groovy e de várias outras ferramentas de desenvolvimento. Para instalar o SDKMAN!, basta usar o seguinte comando:

```
curl -s get.sdkman.io | bash
```

Siga as instruções conforme seu sistema operacional e configurações locais. Depois, execute o seguinte comando:

```
source "$HOME/.sdkman/bin/sdkman-init.sh"
```

Após instalar o SDKMAN!, para instalar a versão mais recente do Groovy e seu *Development Kit*, use o seguinte comando:

```
sdk install groovy
```

Esse comando vai instalar a última versão estável do Groovy, que, no momento da escrita desta obra, era a **4.0.14**. Caso queira instalar outras versões, informe o número da versão ao final do comando. Por exemplo:

```
sdk install groovy 4.0.14
```

Ao final da instalação, o script perguntará se deseja que a versão recém-instalada seja a padrão. Essa decisão fica a seu critério. A instalação de uma versão não exclui outra já instalada. Caso deseje mudar a versão padrão, execute o comando:

```
sdk default groovy 4.0.13
```

Groovy no Linux e macOS – asdf

Outra ferramenta recomendada para Linux e macOS é a **asdf** (https://asdf-vm.com/), que é uma *Command Line Interface* (CLI) que permite a instalação de Groovy e várias outras ferramentas de desenvolvimento.

Para instalar o *asdf*, é necessário, antes, fazer o download dele, considerando que a versão estável no momento da escrita desta obra é a **0.12.0**:

```
git clone https://github.com/asdf-vm/asdf.git ~/.asdf --branch v0.12.0
```

Siga as instruções conforme seu sistema operacional e configurações locais. Depois, execute o seguinte comando:

```
. "$HOME/.asdf/asdf.sh"
```

Inclua no arquivo `~/.bashrc` (no caso de Linux) ou `~/bash_profile` (no caso de macOS):

```
. "$HOME/.asdf/completions/asdf.bash"
```

Após instalar o *asdf*, para instalar a versão mais recente do Groovy e seu *Development Kit*, primeiro instale o plugin Groovy:

```
asdf plugin-add groovy https://github.com/weibemoura/asdf-groovy.git
```

Para instalar a versão mais recente do Groovy, execute este comando:

```
asdf install groovy latest
```

Esse comando vai instalar a última versão estável do Groovy, que, no momento da escrita desta obra, era a **4.0.14**. Caso queira instalar outras versões, informe o número da versão ao final do comando. Por exemplo:

```
asdf install groovy 4.0.14
```

Mesmo que nenhuma versão do Groovy tenha sido instalada antes, deve-se indicar qual a versão padrão (que ele chama de **global**), com este comando:

```
asdf global groovy 4.0.14
```

Ou apenas dizer que se deve usar a versão mais recente que está no *asdf* local:

```
asdf global groovy latest
```

Um diferencial do *asdf* é que cada pasta pode ter uma versão diferente de cada ferramenta. Basta que a pasta tenha um arquivo chamado `.tool-versions` com um conteúdo como este:

```
groovy 4.0.14
```

Por exemplo, se você tem as versões `3.0.19` e `4.0.13` do Groovy no *asdf* do seu computador e definiu que a versão padrão (*global*) é a `3.0.19`, em todo lugar que executar o **Groovy**, será executada a versão `3.0.19`. Porém, na pasta `~/meu-projeto-loko` você criou um arquivo `.tool-versions` com este conteúdo:

```
groovy 4.0.13
```

Nesse cenário, apenas na pasta `~/meu-projeto-loko` a versão do Groovy será a `4.0.13`.

Groovy no Windows – Instalador

Para Windows, a instalação recomendada é pelo uso de um instalador padrão ("assistente" de instalação visual), encontrado em: https://bintray.com/groovy/Distributions/Windows-Installer/view#files/.

Comandos Groovy após sua instalação

Após a instalação do Groovy, seja via **SDKMAN!**, **asdf** (Linux e macOS) ou instalador (Windows), o *PATH* é configurado e os seguintes comandos ficam disponíveis no sistema operacional:

- `groovy` : Comando para execução de arquivos `.groovy` ;
- `groovyc` : Compilador;

- `groovysh` : Terminal Groovy;
- `groovyConsole` : IDE minimalista da linguagem Groovy.

Para que um projeto que usa a linguagem Java ou qualquer outra linguagem da plataforma Java possa usar o Spock para a criação de testes automatizados, também é necessário configurar algumas dependências **Maven**/**Gradle**, conforme a tecnologia de gerenciamento de dependências de seu projeto. Todas as versões do Spock estão nos principais repositórios públicos de bibliotecas.

A seguir, as configurações para os gerenciadores de dependência **Maven** e **Gradle**.

2.3 Configurando as dependências Spock do projeto com Maven

As configurações Maven podem variar caso deseje usar o Spock em conjunto com JUnit 4 ou 5. Porém, algumas configurações são comuns.

Maven – Plugins

Na tag `<build>` , pelo menos um `<plugin>` é necessário para que as classes de teste *groovy* sejam compiladas antes da execução dos testes.

```
<build>
    <plugins>
        <plugin>
            <groupId>org.codehaus.gmavenplus</groupId>
            <artifactId>gmavenplus-plugin</artifactId>
            <version>2.0.0</version>
            <executions>
                <execution>
                    <goals>
```

```
                        <goal>compile</goal>
                        <goal>compileTests</goal>
                    </goals>
                </execution>
            </executions>
        </plugin>
    </plugins>
</build>
```

Outro `<plugin>` dentro de `<plugins>` pode ser necessário caso deseje que suas classes de teste Spock tenham a mesma convenção de nomes das classes JUnit, ou seja, terminando em `Test`. É que, por padrão, as classes de teste Spock terminam com `Spec`.

```
<plugin>
    <artifactId>maven-surefire-plugin</artifactId>
    <version>3.0.0</version>
    <configuration>
        <useModulePath>false</useModulePath>
        <useFile>false</useFile>
        <includes>
            <include>**/*Test</include>
            <include>**/*Spec</include>
        </includes>
        <statelessTestsetReporter
implementation="org.apache.maven.plugin.surefire.extensions.junit5.JUnit5X
ml30StatelessReporter">
            <disable>false</disable>
            <version>3.0</version>
            <usePhrasedFileName>false</usePhrasedFileName>

<usePhrasedTestSuiteClassName>true</usePhrasedTestSuiteClassName>

<usePhrasedTestCaseClassName>true</usePhrasedTestCaseClassName>

<usePhrasedTestCaseMethodName>true</usePhrasedTestCaseMethodName>
        </statelessTestsetReporter>
    </configuration>
</plugin>
```

Maven – Dependência geral do Spock

A dependência geral do Spock deve ser adicionada conforme o código a seguir, após a tag `<build>` :

```
<dependencyManagement>
    <dependencies>
        <dependency>
            <groupId>org.spockframework</groupId>
            <artifactId>spock-bom</artifactId>
            <version>2.3-groovy-4.0</version>
            <type>pom</type>
            <scope>import</scope>
        </dependency>
    </dependencies>
</dependencyManagement>
```

Maven – Dependências do Spock e Groovy

As dependências do Spock e do Groovy devem ser adicionadas dentro de `<dependencies>` , logo após a configuração anterior.

```
<dependency>
    <groupId>org.spockframework</groupId>
    <artifactId>spock-core</artifactId>
    <scope>test</scope>
</dependency>

<dependency>
    <groupId>org.apache.groovy</groupId>
    <artifactId>groovy</artifactId>
    <version>4.0.14</version> <!-- ou qualquer versão da família 4.0.x -->
    <scope>test</scope>
</dependency>
```

Maven – Dependências exclusivas para uso do JUnit 5

Caso deseje usar o Spock e testes escritos em JUnit 5, são necessárias mais duas dependências dentro de `<dependencies>` .

```
<dependency>
    <groupId>org.junit.jupiter</groupId>
    <artifactId>junit-jupiter-api</artifactId>
    <version>5.9.3</version> <!-- ou qualquer versão da família até 5.9.x
-->
    <scope>test</scope>
</dependency>
<dependency>
    <groupId>org.junit.jupiter</groupId>
    <artifactId>junit-jupiter-engine</artifactId>
    <version>5.9.3</version>
    <scope>test</scope> <!-- ou qualquer versão da família até 5.9.x, mas
use a mesma do JUnit -->
</dependency>
```

> **Atenção!** Até o momento da escrita deste livro, versões `5.10.x` do JUnit 5 tinham alguns problemas de compatibilidade com o IntelliJ.

Maven – Dependências exclusivas para uso do JUnit 4

Caso deseje usar o Spock e testes escritos em JUnit 4, são necessárias mais duas dependências dentro de `<dependencies>`.

```
<dependency>
    <groupId>org.spockframework</groupId>
    <artifactId>spock-junit4</artifactId>
    <scope>test</scope>
</dependency>
<dependency>
    <groupId>org.junit.vintage</groupId>
    <artifactId>junit-vintage-engine</artifactId>
    <version>5.9.3</version> <!-- ou qualquer versão da família até 5.9.x
-->
</dependency>
```

Não é necessário adicionar a dependência do JUnit 4 em `<dependencies>`.

2.4 Configurando as dependências Spock do projeto com Gradle

Se o seu projeto usar o Gradle como gerenciador de dependências, é necessário apenas configurar **um *script plugin*** e **algumas dependências**.

Plugin `groovy`

O ***script plugin*** que deve ser configurado no Gradle é o **groovy**, cuja configuração é demonstrada no código a seguir:

```
apply plugin: 'groovy'
```

Caso já tenha o bloco `plugins`, basta adicionar o plugin `groovy`:

```
plugins {
  id 'java'
  id 'groovy'
}
```

Gradle – Dependências do Spock

No código do exemplo a seguir, é mostrado como incluir a **dependência mínima** da versão do Spock framework usada nesta obra:

```
testImplementation platform("org.spockframework:spock-bom:2.3-groovy-4.0")
testImplementation "org.spockframework:spock-core"
```

Gradle – Dependência do Groovy

Devemos incluir uma dependência específica do **Groovy**, como no próximo código:

```
implementation platform('org.apache.groovy:groovy-bom:4.0.14')
implementation 'org.apache.groovy:groovy'
```

Gradle – Dependências exclusivas para uso do JUnit 5

Caso deseje usar o Spock e testes escritos em JUnit 5, são necessárias mais duas dependências dentro de `dependencies`.

```
testImplementation 'org.junit.jupiter:junit-jupiter-engine:5.9.3'
testImplementation 'org.junit.jupiter:junit-jupiter-api:5.9.3'
```

> **ATENÇÃO!** Até o momento da escrita deste livro, versões `5.10.x` do JUnit 5 tinham alguns problemas de compatibilidade com o IntelliJ.

Gradle – Dependências exclusivas para uso do JUnit 4

Caso deseje usar o Spock e testes escritos em JUnit 4, são necessárias mais duas dependências dentro de `dependencies`.

```
testImplementation 'org.spockframework:spock-junit4'
testImplementation 'org.junit.vintage:junit-vintage-engine:5.9.3'
```

Conclusão

Neste capítulo, foram explicadas e exemplificadas quais configurações são necessárias para começar os trabalhos com Spock framework. No próximo capítulo, começaremos a criar e executar nossos **primeiros testes**.

Capítulo 3
Primeiros testes automatizados com Spock

No capítulo anterior, vimos como preparar o ambiente para trabalhar com Spock. Com o ambiente configurado e testado, podemos começar a criar nossos primeiros testes com Spock. Neste capítulo, veremos onde criá-los e as formas de execução dos testes. As saídas geradas pelos resultados dos testes também serão explicadas.

3.1 Primeiro teste: A classe Math calcula potência corretamente?

Se tiver criado um projeto Maven ou Gradle, basta criar suas classes Groovy de teste no diretório `src/test/groovy`. É nele que criaremos nosso primeiro teste com Spock.

Nosso primeiro teste vai testar a classe `Math` do Java, para verificar se seu método `pow()` calcula corretamente uma potência. É claro que esse método da API do Java funciona e não precisaríamos testá-lo. Esse primeiro exemplo é apenas para que nosso primeiro contato com o Spock seja rápido e de fácil compreensão. Vejamos o código a seguir, que pode ser criado em um arquivo chamado `MathTest.groovy`, já que faremos testes para a classe `Math`:

```
package livrospock.capitulo03
import spock.lang.Specification

class MathTest extends Specification {
    def '2 ao quadrado deve ser 4'() {
        expect:
        Math.pow(2,2) == 4
    }
}
```

Nesse código, note que usamos a declaração de `package` (pacote) do Java. O pacote dos testes Spock pode ser qualquer um e, assim como em Java, apenas indica o caminho relativo de diretórios onde o arquivo está no diretório de códigos-fontes do projeto. Apenas para deixar claro, quando estamos descrevendo códigos de testes ou não, os de teste estarão sempre em subpacotes de `unit.`.

O próximo detalhe a destacar é que a classe de teste é uma subclasse de `spock.lang.Specification`. **Todo teste com Spock deve ser uma subclasse de Specification** para que funcione com todas as funcionalidades do framework.

Note ainda que a classe não é `public`. Na verdade, ela **é** pública. É que o termo `public` é opcional em Groovy e, se omitido, significa que a classe será pública em tempo de execução.

Algo que pode parecer estranho para quem nunca teve contato com Groovy é que o nome do método é uma **frase entre aspas**. Isso é permitido em Groovy e facilita muito a criação de testes, afinal, cada método de teste em uma classe de testes automatizados deveria executar um **cenário de teste**. Qual nome de método deixa mais claro o cenário a ser testado: `def doisAoQuadradoDeveSerQuatro()` ou `def '2 ao quadrado deve ser 4'()`? Essa sintaxe é muito usada em testes com Spock, sendo predominante na documentação oficial.

O bloco `expect:` indica onde estão as asserções (ou verificações) que determinam o resultado do teste. Sem ele, o teste não teria sido executado. Outra característica muito importante é que as asserções em Spock normalmente contêm literalmente testes lógicos e não métodos, como se faz em **JUnit**. No exemplo há somente uma asserção, porém podem haver quantas forem necessárias e o teste só passará se todas passarem. O exemplo a seguir demonstra como seria um teste que faz mais de uma asserção nesse bloco:

```
class MathTest extends Specification {
    def 'sqrt deve calcular a raiz quadrada'() {
        expect:
        Math.sqrt(4) == 2
```

```
        Math.sqrt(25) == 5
        Math.sqrt(144) == 12
    }
}
```

Nesse exemplo, queríamos saber se o método `sqrt` da classe `Math` realmente calcula a raiz quadrada corretamente e, para isso, fizemos três asserções no bloco `expect`.

> Caso você tenha pouco conhecimento sobre Groovy, o *Apêndice A – Guia de Groovy para desenvolvedores Java* possui um conteúdo que dá o *caminho das pedras* dessa linguagem.

3.2 Executando um teste

Para executar **todas** as classes de teste de um projeto, basta executar o comando de seu gerenciador de dependências que invoca os testes do projeto. No Maven, por exemplo, todos os *goals* menos o `compile` promovem a execução dos testes do projeto.

A execução de apenas uma classe de testes pode ser feita usando o comando específico de seu gerenciador de dependência recomendado para isso. No Maven, o comando seria como o do exemplo a seguir:

```
mvn -Dtest=MathTest test
```

No Gradle, o comando seria como este:

```
gradle -Dtest.single=MathTest test
```

Caso use as IDEs IntelliJ ou Eclipse, será possível executar a classe `MathTest` como se fosse uma classe **JUnit**. Ou seja, basta pedir a execução da classe que o teste unitário nela será executado. Essas IDEs permitem ainda a execução de apenas um dos métodos na

classe de teste. Caso utilize a IDE Netbeans ou editores simples (VS Code, Sublime Text, Vim etc.), sua única alternativa será executar o comando de execução individual de testes de seu gerenciador de dependências, conforme os três últimos códigos de exemplo.

Caso o teste compile e seja executado, ele gerará uma saída no log de execução como esta:

```
-------------------------------------------------------
 T E S T S
-------------------------------------------------------
Running livrospock.capitulo03.MathTest
Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.156 sec

Results :

Tests run: 1, Failures: 0, Errors: 0, Skipped: 0
```

3.3 O que acontece quando um teste falha

Um teste pode falhar por exceção não esperada em tempo de execução ou porque ao menos uma das asserções nele não passou. Vejamos o que ocorre em cada uma dessas situações.

Teste que falha por exceção

Tomemos como exemplo o código a seguir, que possui uma sutil diferença em relação ao anterior, na qual trocamos a base por uma divisão entre 1 e 0:

```
package livrospock.capitulo03
import spock.lang.Specification

class MathTest extends Specification {
    def '2 ao quadrado deve ser 4'() {
        expect:
```

```
        Math.pow((1/0), 2) == 4
    }
}
```

Esse código vai compilar, mas provocará uma exceção em tempo de execução, pois `1/0` em Groovy, assim como em Java, resulta em uma `java.lang.ArithmeticException: Division by zero`. Ao ser executado, o teste geraria uma saída no log como esta para esse exemplo:

```
2 ao quadrado deve ser 4(MathTest)  Time elapsed: 0.054 sec  <<< FAILURE!
Condition failed with Exception:

Math.pow((1/0), 2) == 5
           |
           java.lang.ArithmeticException: Division by zero

    at MathTest.2 ao quadrado deve ser 4(MathTest.groovy:6)
Caused by: java.lang.ArithmeticException: Division by zero
    at java.math.BigDecimal.divide(BigDecimal.java:1742)
```

Como é possível notar, o log de erro do Spock **indica o local exato que provocou a exceção**. Com outras ferramentas de testes, saberíamos a linha, mas não o ponto exato. Note ainda como o log fica mais fácil de ser lido, pois o nome do método é, literalmente, o nome do cenário.

O formato e os detalhes da saída gerada pelo Spock ajudam muito naquelas situações em que ocorre o clássico `NullPointerException`. Vamos simular esse tipo de exceção a partir do código de teste a seguir, no qual declaramos um `Double base = null` propositalmente para que a invocação de seu `.intValue()` provoque uma exceção:

```
package livrospock.capitulo03
import spock.lang.Specification

class MathTest extends Specification {
    def '2 ao quadrado deve ser 4'() {
        Double base = null
```

```
        expect:
        Math.pow(base.intValue(), 2) == 4
    }
}
```

Ao ser executado, esse teste resultaria em um log como este:

```
2 ao quadrado deve ser 4(MathTest)  Time elapsed: 0.058 sec  <<< FAILURE!
Condition failed with Exception:

Math.pow(base.intValue(), 2) == 4
           |
           java.lang.NullPointerException: Cannot invoke method intValue()
on null object

    at MathTest.2 ao quadrado deve ser 4(MathTest.groovy:7)
Caused by: java.lang.NullPointerException: Cannot invoke method intValue()
on null object
    at java.math.BigDecimal.divide(BigDecimal.java:1742)
```

Note que o exato objeto que provocou o `NullPointerException` é indicado, evitando ajustes e/ou novas execuções para descobrir quem é o culpado pela exceção.

Como foi demonstrado nos exemplos, o ponto específico onde ocorreu a exceção ficaria indicado, o que facilita muito o tratamento do problema. Com outras ferramentas, teríamos que executar várias vezes, mexendo um pouco no código (*tentativa e erro*), ou teríamos que executar o teste em *modo debug* caso a IDE usada possua esse recurso.

Teste que não passa em uma asserção

Comparação entre números

Tomemos como exemplo o código a seguir, que é muito parecido com o anterior, porém contém um erro proposital no resultado esperado (5 em vez de 4 como resultado esperado para 2 ao quadrado):

```
package livrospock.capitulo03
import spock.lang.Specification

class MathTest extends Specification {
    def '2 ao quadrado deve ser 4'() {
        expect:
        Math.pow(2, 2) == 5
    }
}
```

Esse código vai compilar e executar sem exceção alguma, porém a asserção (ou verificação) configurada no bloco `expect` vai **falhar**, pois 2 ao quadrado não é igual a 5. Ao ser executado, o teste geraria uma saída no log como a do exemplo a seguir:

```
Running livrospock.capitulo03.MathTest
Tests run: 1, Failures: 1, Errors: 0, Skipped: 0, Time elapsed: 0.185 sec
<<< FAILURE!
2 ao quadrado deve ser 4(MathTest)  Time elapsed: 0.059 sec  <<< FAILURE!
Condition not satisfied:

Math.pow(2, 2) == 5
      |        |
      4.0      false

    at MathTest.2 ao quadrado deve ser 4(MathTest.groovy:8)
```

Como é possível notar novamente, **o log de erro do Spock indica o local exato da falha de verificação e os valores envolvidos no teste**. Ficaram claros não só a linha da falha, mas também os valores comparados. Note que o resultado de `pow(2,2)` ficou explícito no log. Isso nos poupa de criar uma mensagem de erro recuperando, manualmente, o resultado do método.

Comparação entre alfanuméricos

Quando criamos testes que comparam valores alfanuméricos (`String`), além de apontar que existe a diferença, o Spock gera um pequeno relatório estatístico das diferenças. Para ver como isso

funciona, vamos criar outra classe de teste, a `StringTest`. Nela, vamos testar os métodos `toUpperCase()` e `substring()` da biblioteca padrão do Java.

```
class StringTest extends Specification {

  def "toUpperCase() deveria pôr tudo em caixa alta"() {
    expect:
    "bom dia".toUpperCase() == "Bom Dia"
  }

  def "substring() deveria extrair a parte desejada do texto"() {
    expect:
    "sejam bem vindos!".substring(0,4) == "sejam"
  }
}
```

No último código, montamos dois testes que vão falhar. O objetivo é que você veja como é o log que o Spock gera em caso de falha na comparação entre diferentes objetos `String`. A execução desse teste geraria logs como os dos textos a seguir:

```
toUpperCase() deveria por tudo em caixa
alta(livrospock.capitulo03.StringTest)  Time elapsed: 0.208 sec  <<<
FAILURE!
org.spockframework.runtime.SpockComparisonFailure: Condition not
satisfied:

"bom dia".toUpperCase() == "Bom Dia"
          |             |
          BOM DIA       false
                        4 differences (42% similarity)
                        B(OM) D(IA)
                        B(om) D(ia)

substring() deveria extrair a parte desejada do
texto(livrospock.capitulo03.StringTest)  Time elapsed: 0.006 sec  <<<
FAILURE!
org.spockframework.runtime.SpockComparisonFailure: Condition not
satisfied:
```

```
"sejam bem vindos!".substring(0,4) == "sejam"
                   |              |
                   seja           false
                                  1 difference (80% similarity)
                                  seja(-)
                                  seja(m)
```

Note nos dois logs anteriores que, além de serem indicados o local exato da diferença e quais os valores esperado e recuperado, são descritos uma porcentagem de semelhança e os detalhes de cada posição no texto onde foram encontradas diferenças.

Comparação entre coleções

Ao comparar o conteúdo ou o tamanho de coleções, o Spock imprime no log não só a linha e o ponto da diferença como os conteúdos das coleções envolvidas. Vamos criar mais um teste, o `CollectionsTest` (veja o código-fonte a seguir), no qual teremos três testes entre coleções. Caso fique com dúvidas sobre a lista e/ou o mapa criados com Groovy, não se esqueça de que pode recorrer ao *Apêndice A – Guia de Groovy para desenvolvedores Java*.

```
class CollectionsTest extends Specification {

    def 'listas devem ser iguais'() {
        expect:
        def lista1 = ['café','leite','açúcar']
        def lista2 = ['café','leite','adoçante']
        lista1 == lista2
    }

    def 'mapas devem ser iguais'() {
        expect:
        def mapa1 = [PA:'Pará', SP:'São Paulo', MG:'Minas Gerais']
        def mapa2 = [DF:'Distrito Federal', GO:'Goiás', PR:'Paraná']
        mapa1 == mapa2
    }
```

```
    def 'listas devem ter o mesmo tamanho'() {
        expect:
        def lista1 = ['uva','morango','amora']
        def lista2 = ['abacate','maçã verde']
        lista1.size() == lista2.size()
    }
}
```

No primeiro teste, comparamos o conteúdo de duas listas (`List`). No segundo, comparamos o conteúdo de dois mapas (`Map`). Já no terceiro teste, comparamos o tamanho de duas listas (`List`). Todos eles vão falhar. O log produzido será como o do exemplo a seguir:

```
listas devem ser iguais (...) Condition not satisfied:

lista1 == lista2
|      |  |
|      |  [café, leite, adoçante]
|      false
[café, leite, açúcar]


mapas devem ser iguais (...) Condition not satisfied:

mapa1 == mapa2
|     |  |
|     |  [DF:Distrito Federal, GO:Goiás, PR:Paraná]
|     false
[PA:Pará, SP:São Paulo, MG:Minas Gerais]


listas devem ter o mesmo tamanho (...) Condition not satisfied:

lista1.size() == lista2.size()
|      |      |  |      |
|      3      |  |      2
|             |  [abacate, maçã verde]
|             false
[uva, morango, amora]
```

Note que, no log desses três testes com coleções, além de indicar o ponto exato da falha do teste, o Spock imprimiu os conteúdos das coleções comparadas.

Conforme demonstram os exemplos que acabamos de ver, independente do que estamos comparando, o Spock nos fornece muitos detalhes. Isso reduz muito a necessidade de *tentativa e erro* e de execução dos testes em *modo Debug* da IDE para identificar e corrigir o problema.

O Spock ajuda até quando a escrita do teste está errada

Nos exemplos que acabamos de ver, os testes foram escritos corretamente. Porém, como somos humanos, ficamos cansados, estressados, tristes, aborrecidos etc. Logo, é bem provável que nos esqueçamos de algum detalhe ao escrever um teste. Porém, dependendo do detalhe, o Spock nos dá dicas de como consertar nosso código de teste. Vejamos o exemplo a seguir:

```
package livrospock.capitulo03
import spock.lang.Specification

class MathTest extends Specification {
    def '2 ao quadrado deve ser 4'() {
        setup:
        def esperado = 4

        expect:
        esperado = Math.pow(2, 2)
    }
}
```

Notou que, no bloco `expect`, temos uma atribuição (`=`) em vez de uma comparação? Veja como seria o log do Spock na execução desse teste:

```
LutadorServiceTest.groovy:10:9
Groovyc: Expected a condition, but found an assignment. Did you intend to
write '==' ?
```

O log diz que na linha 10, coluna 9, do arquivo de teste houve o seguinte erro: *Esperava-se uma condição, mas há uma atribuição*. Então segue uma sugestão no log sobre usar `==`.

Vejamos outro exemplo de código de teste com erros:

```
package livrospock.capitulo03
import spock.lang.Specification

class MathTest extends Specification {
    def '2 ao quadrado deve ser 4'() {
        setup:
        def esperado = 4

        expect:
        esperado = Math.poe(2, 2)
    }
}
```

Note que, em vez de `pow`, está `poe`. Esse erro é bem provável, uma vez que `E` fica ao lado de `W` na maioria dos teclados. Considerando que Groovy é uma linguagem de script, esse código não ficaria marcado como errado em uma IDE. Porém, ao tentar executá-lo, veja como seria o log do Spock na execução desse teste:

```
Condition failed with Exception:

esperado == Math.poe(2, 2)
|           |    |
4           |    groovy.lang.MissingMethodException: No signature of
method: static java.lang.Math.poe() is applicable for argument types:
(Integer, Integer) values: [2, 2]
            |    Possible solutions: max(int, int), min(int, int),
pow(double, double), any(), abs(int), cos(double)
            |        at livrospock.service.LutadorServiceTest.2 ao
quadrado deve ser 4(LutadorServiceTest.groovy:10)
            class java.lang.Math
```

Observe que o log indica possíveis opções para usar em vez de `poe`: `max(int, int)`, `min(int, int)`, `pow(double, double)`, `any()`, `abs(int)`,

`cos(double)` . Isso ajuda muito na correção do erro, diminuindo as chances de termos que recorrer a consultas em mecanismos de busca ou IAs generativas na internet.

3.4 Forçando a execução de várias asserções (ou verificações)

Podem ocorrer situações onde queremos realizar várias asserções em um mesmo teste. Porém, se o fizermos como foi mostrado até aqui, **o Spock para a execução do teste assim que se deparar com uma asserção que falha**, deixando de executar possíveis próximas asserções. Vejamos um exemplo disso no código a seguir:

```
def 'Math.pow() deve calcular a potência corretamente'() {
    expect:
    Math.pow(2, 2) == 4
    Math.pow(2, 3) == 6
    Math.pow(3, 2) == 9
    Math.pow(3, 3) == 18
}
```

A segunda asserção falhará, pois 2 elevado a 3 é 8 e não 6. Porém, se o teste for executado com esse código, as demais asserções sequer serão executadas. O problema disso é que a última asserção também falhará, pois 3 elevado a 3 é 27 e não 18. Só que só seria possível ver que essa asserção falha após corrigir o erro na segunda. Isso pode ser um processo bem trabalhoso caso tenhamos muitas asserções em um teste.

Para evitar esse tipo de situação e pegar todas as asserções com falha de uma só vez, podemos incluir uma pequena mudança no código do teste, que ficaria como no próximo código:

```
def 'Math.pow() deve calcular a potência corretamente'() {
    expect:
```

```
    verifyAll {
        Math.pow(2, 2) == 4
        Math.pow(2, 3) == 6
        Math.pow(3, 2) == 9
        Math.pow(3, 3) == 18
    }
}
```

O que há de novo no código é que as asserções estão em um bloco `verifyAll{}`. Quando várias asserções estão em um bloco como esse, significa que todas serão executadas, mesmo que uma ou mais falhem durante a asserção (ou verificação). O resultado da execução desse teste seria como o do log a seguir:

```
Condition not satisfied:

Math.pow(2, 3) == 6
|    |         |
|    8.0       false
class java.lang.Math


...

Condition not satisfied:

Math.pow(3, 3) == 18
|    |         |
|    27.0      false
class java.lang.Math
```

Note que ambas as asserções com falha constam na execução.

3.5 Exemplos de execução de testes no IntelliJ e Eclipse

Veja a seguir algumas imagens que mostram como seriam as execuções de testes escritos em Spock nas IDEs IntelliJ e Eclipse. A seta de cima aponta a opção para executar todos os testes da classe. Nas setas de baixo, temos como pedir a execução do teste de um método apenas.

Execução no IntelliJ 2023.2

Execução a partir de uma classe (todos os testes ou apenas um deles):

Figura 3.1: Execução de testes Spock de uma classe ou método no IntelliJ

Execução a partir de um pacote:

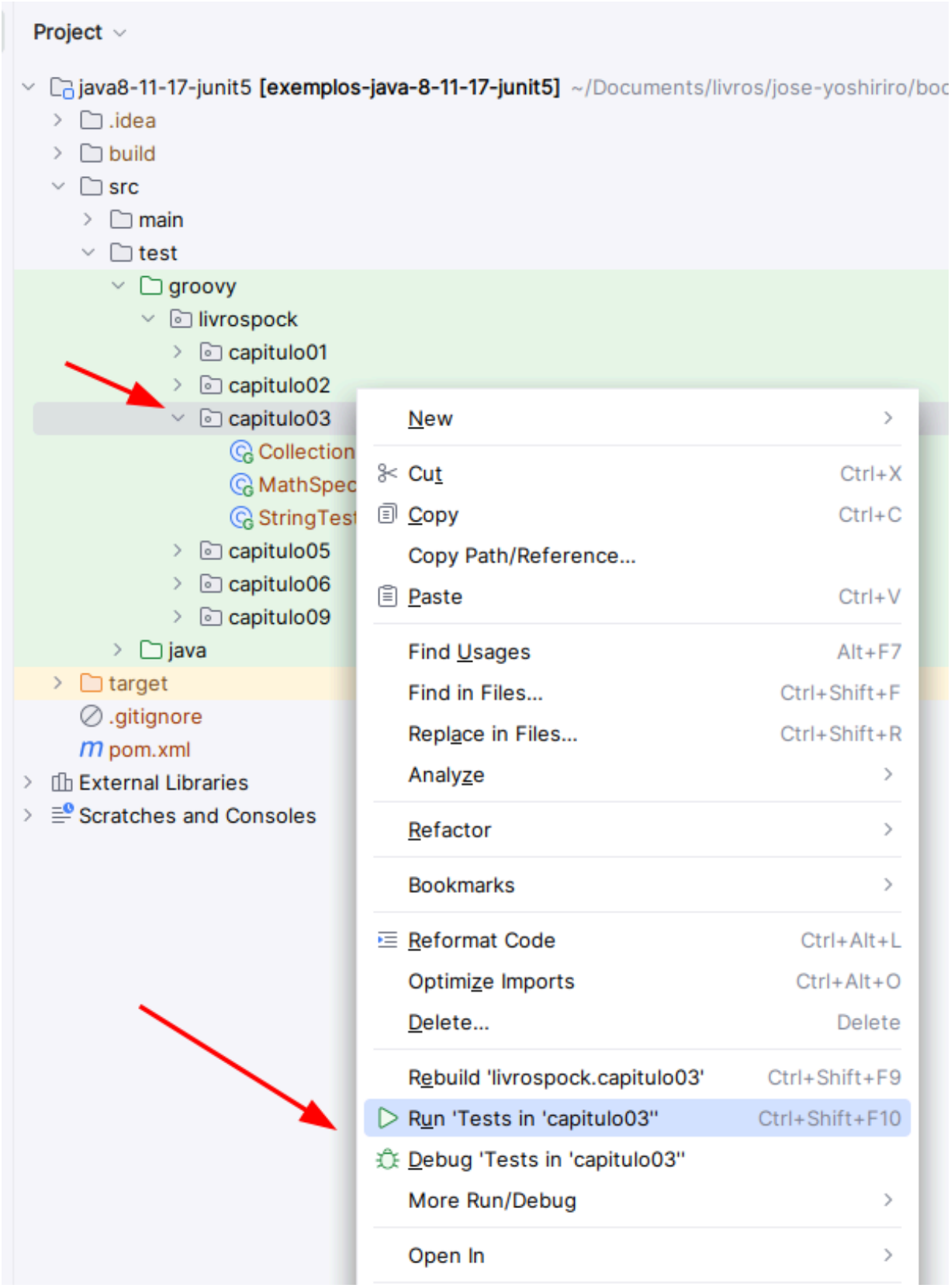

Project
java8-11-17-junit5 [exemplos-java-8-11-17-junit5] ~/Documents/livros/jose-yoshiriro/boo
.idea
build
src
main
test
groovy
livrospock
capitulo01
capitulo02
capitulo03
Collection
MathSpec
StringTest
capitulo05
capitulo06
capitulo09
java
target
.gitignore
pom.xml
External Libraries
Scratches and Consoles
New
Cut Ctrl+X
Copy Ctrl+C
Copy Path/Reference...
Paste Ctrl+V
Find Usages Alt+F7
Find in Files... Ctrl+Shift+F
Replace in Files... Ctrl+Shift+R
Analyze
Refactor
Bookmarks
Reformat Code Ctrl+Alt+L
Optimize Imports Ctrl+Alt+O
Delete... Delete
Rebuild 'livrospock.capitulo03' Ctrl+Shift+F9
Run 'Tests in 'capitulo03'' Ctrl+Shift+F10
Debug 'Tests in 'capitulo03''
More Run/Debug
Open In

Figura 3.2: Execução de testes Spock de um pacote no IntelliJ

Resultado da execução de testes:

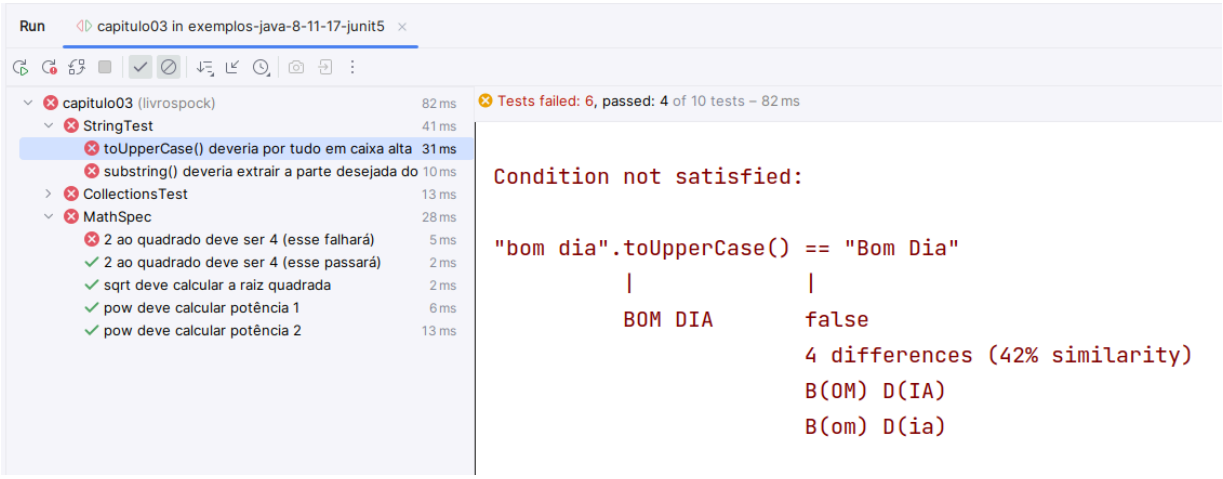


Figura 3.3: Resultado da execução de testes Spock no IntelliJ

Execução no Eclipse 2023-06

Execução a partir de uma classe:

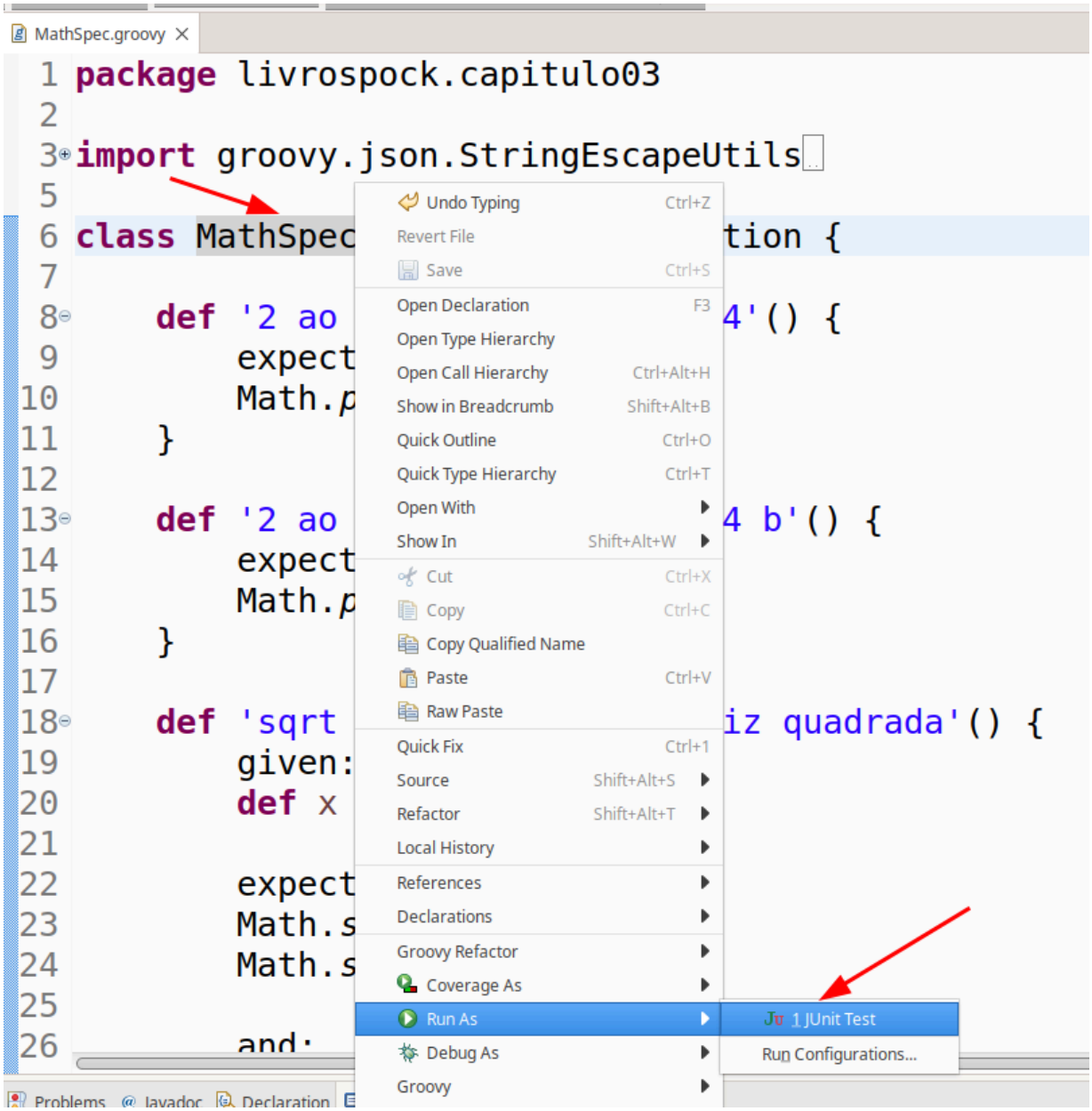


Figura 3.4: Execução de testes Spock de uma classe no Eclipse

Execução a partir de um método de teste:

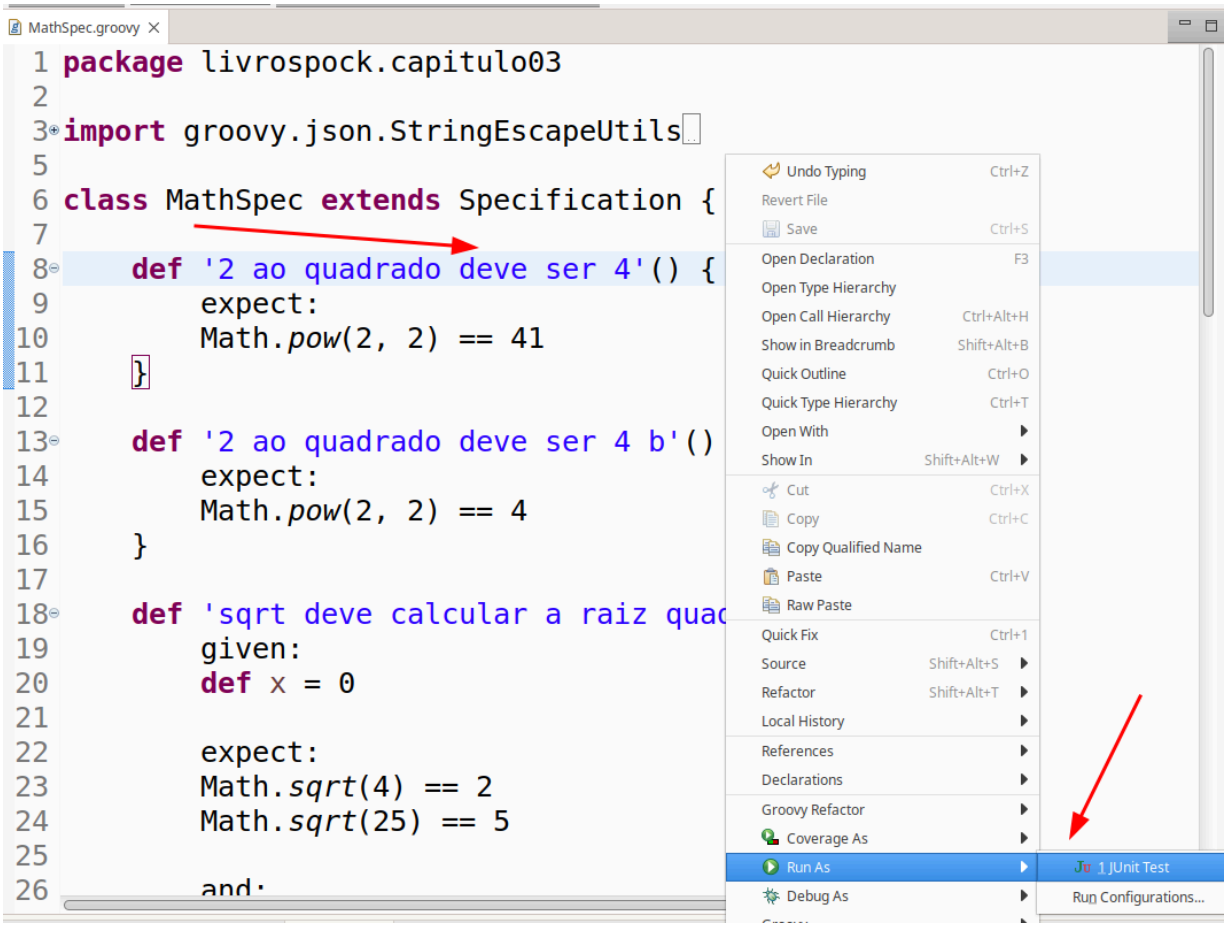


Figura 3.5: Execução de testes Spock de um método no Eclipse

Execução a partir de um pacote:

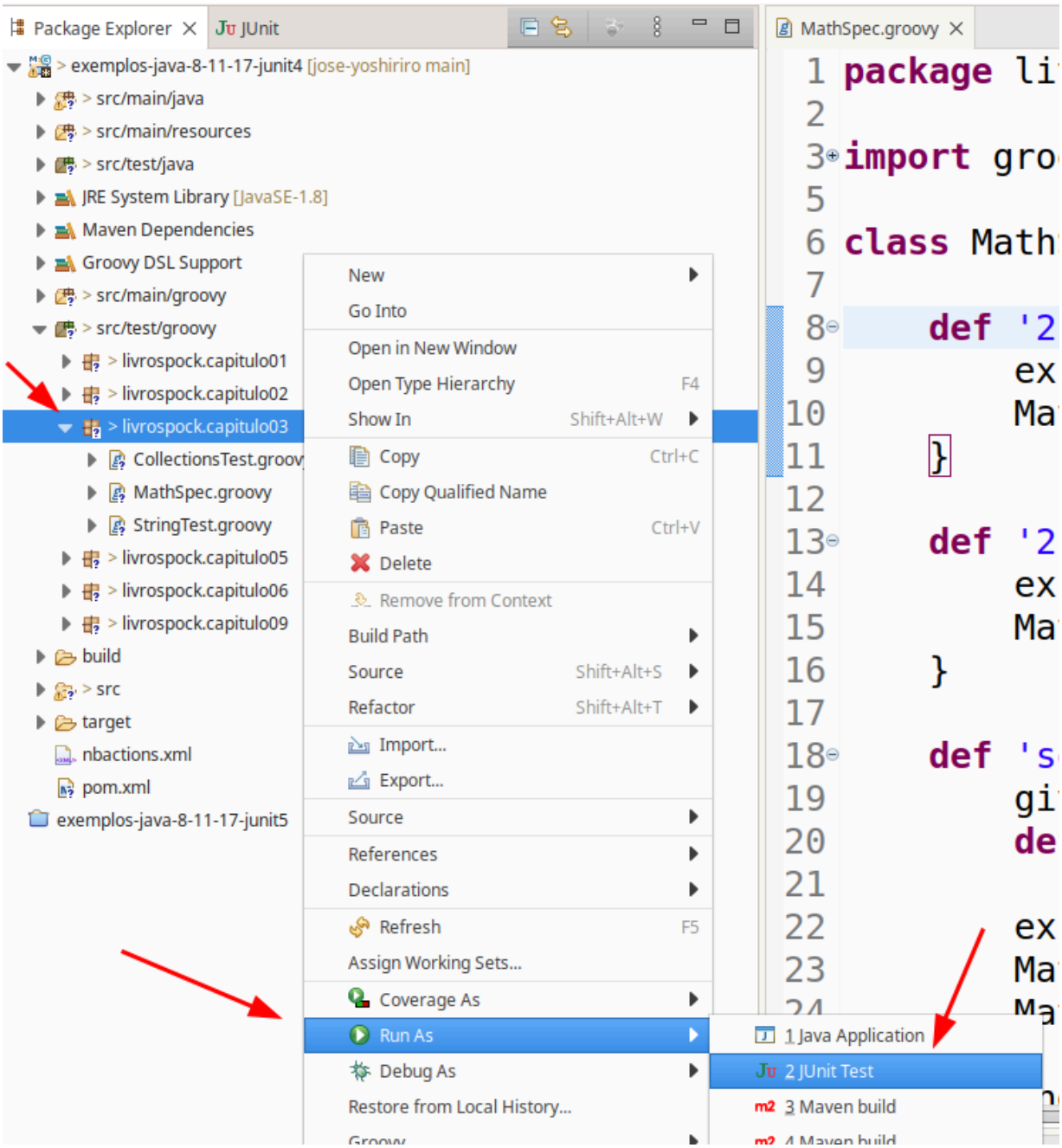


Figura 3.6: Execução de testes Spock a partir de um pacote no Eclipse

Resultado da execução de testes:

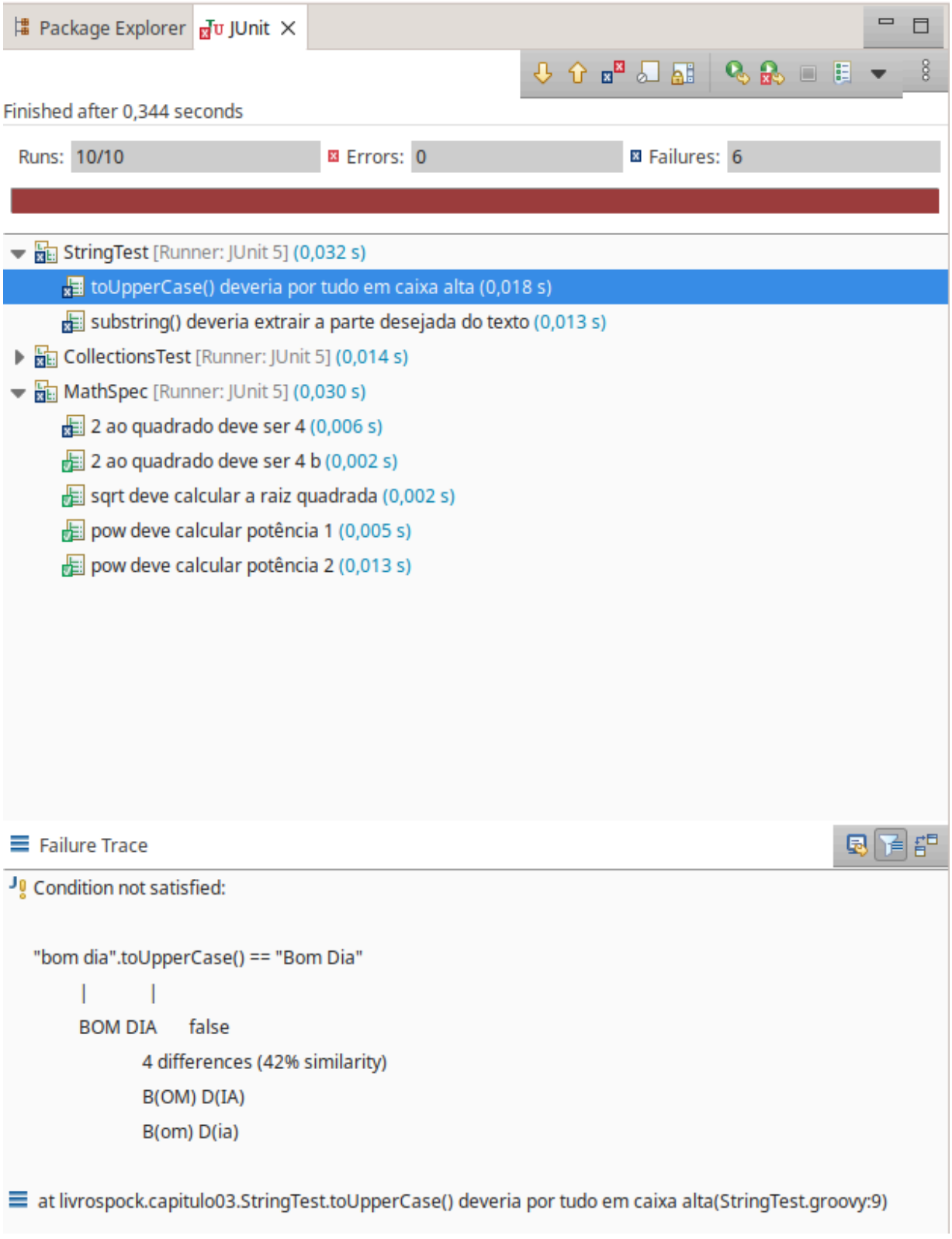


Figura 3.7: Resultado da execução de testes Spock no Eclipse

3.6 Problema na execução de testes no Eclipse 2018 e 2019

Versões do Eclipse lançadas em 2018 e 2019 adotavam o **JUnit 5** como *Test Runner* padrão. Isso, a princípio, impossibilita a execução de testes Spock diretamente pela IDE. Infelizmente, o Eclipse não permite alterar o *Test Runner* padrão nem da IDE nem por projeto, já que mesmo a versão mais recente do Spock até o fechamento desta obra usa o **JUnit 4**. Felizmente, é possível contornar isso, embora exija uma pequena configuração logo após a primeira execução individual de cada teste. A seguir, os passos para a solução de contorno:

- **Passo 1: Execute o teste pelo assistente do Eclipse**. Solicite a execução da classe ou método com `Run as... > JUnit Test`. A execução vai falhar com uma mensagem sobre o JUnit 5.
- **Passo 2: Ajuste o *Test Runner* para JUnit 4**. Após a tentativa de execução, o Eclipse cria uma *Run Configuration* para a execução da classe ou método. Vá ao formulário de edição das *Run Configurations* e encontre a que corresponde à execução do teste desejado. Altere então seu *Test Runner* para **JUnit 4** (veja a figura a seguir).

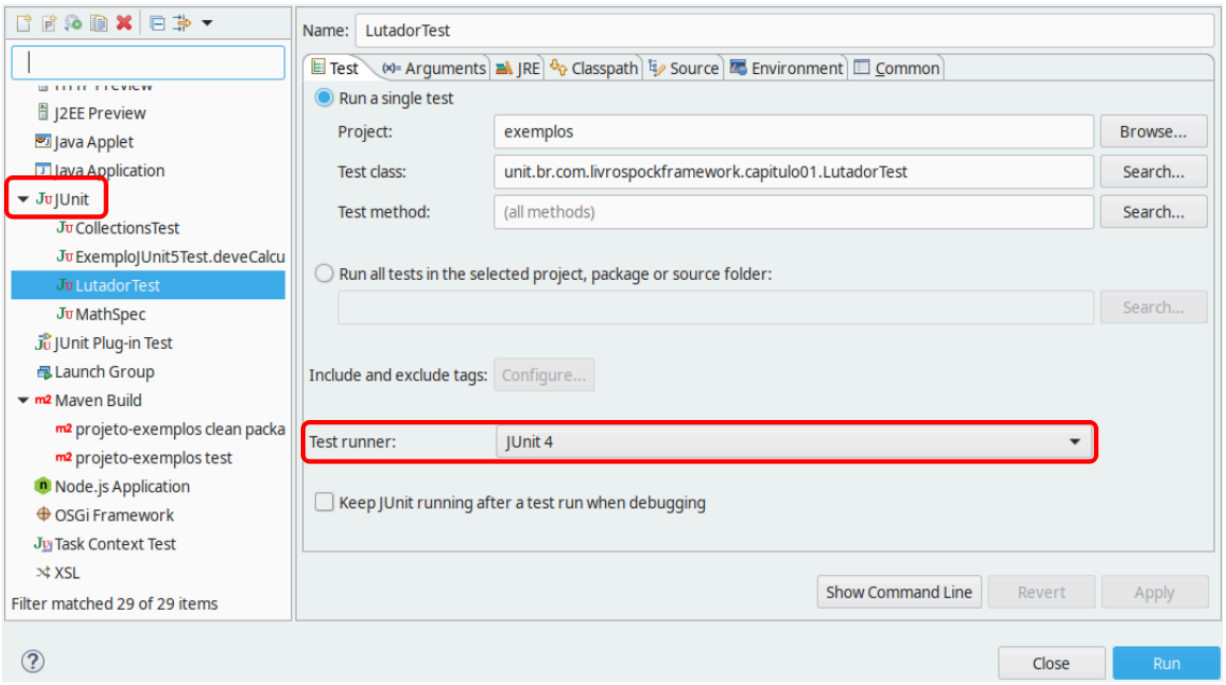


Figura 3.8: Alteração do Test Runner para JUnit 4 no Eclipse

Após a execução desses dois passos, sempre que solicitar a execução novamente, o teste será realizado sem problemas e com a exibição atualizada do painel de testes JUnit do Eclipse.

Versões a partir de **2020** já não apresentam esse problema.

Conclusão

Neste capítulo, criamos nossos primeiros testes com Spock. Vimos como criar um teste simples e as formas de executá-lo. Também vimos como o Spock registra os testes em caso de exceção ou falha de verificação. No próximo capítulo, veremos com mais detalhes a **anatomia dos testes** criados com esse framework.

Capítulo 4
Anatomia de um teste Spock

No capítulo anterior, criamos nossos primeiros testes com o Spock e vimos como são as saídas produzidas em caso de sucesso e falha na execução.

Neste capítulo, veremos os conceitos e convenções do framework Spock para que possamos compreender melhor a anatomia dos testes criados com essa ferramenta. Isso será muito importante para a criação de testes melhores e com código menor e mais organizado.

4.1 Testes são chamados de Specifications

Uma classe de teste no Spock é chamada em sua documentação oficial de ***Specification***. Por isso, ela apresenta as classes de teste com o sufixo `Spec` no nome. A classe `MathTest` criada no capítulo anterior, por exemplo, poderia ter o nome alterado para `MathSpec`, se fôssemos seguir a convenção da documentação do Spock.

Criar as Specifications com o sufixo *Test* possui algum efeito colateral?

Não. Inclusive todos os testes neste livro terão o sufixo `Test` no nome. A razão disso é que, além de continuarem funcionando perfeitamente, seguem a convenção do Maven e afins, que executam os testes em classes que terminam com esse termo e estão no diretório `src/test`.

E se eu preferir usar o sufixo *Spec* nos nomes de meus testes?

Você perderá a execução automática dos testes pelo Maven, pois ele não vai considerar essas classes como testes a serem executados, mesmo se estiverem no diretório `src/test`. Para que executem os testes em classes com o sufixo `Spec`, é necessário adicionar uma configuração apropriada para o gerenciador de dependências que estiver usando. A seguir, exemplos de como configurar isso no Maven. No Gradle, essa configuração não é necessária.

Configurando o Maven para compilar e executar testes com o sufixo `Spec`:

```
<plugin>
    <artifactId>maven-surefire-plugin</artifactId>
    <version>2.18.1</version> <!-- ou versão mais recente -->
    <configuration>
      <includes>
        <include>**/*Spec.groovy</include>
        <include>**/*Spec.java</include>
        <include>**/*Test.groovy</include>
        <include>**/*Test.java</include>
      </includes>
    </configuration>
</plugin>
```

Note que tivemos que indicar que arquivos `.groovy` e `.java` com os sufixos `Spec` e `Test` devem ser considerados classes de teste. É preciso incluir todos esses 4 padrões, caso contrário, se o projeto já possuir testes legados em **JUnit**, eles deixariam de ser executados.

Toda classe de teste deve ser subclasse de `spock.lang.Specification`

Nos exemplos abordados no capítulo anterior vimos isso e, como já foi dito lá, essa herança é realmente **necessária**. As classes de testes podem estender diretamente ou indiretamente (ou seja,

estender outra que estenda) a `spock.lang.Specification`. Ao estender essa classe, todos os recursos do framework Spock ficam disponíveis para a classe de teste.

4.2 Uma classe de teste pode conter vários métodos de teste

Em geral costuma-se criar uma classe de testes para cada classe no projeto. E, para cada método da classe testada, criam-se vários métodos de teste, de acordo com a quantidade de cenários necessários. O Spock permite a criação de quantos métodos de teste forem necessários na mesma classe de teste. Um exemplo desse tipo de situação está na figura a seguir:

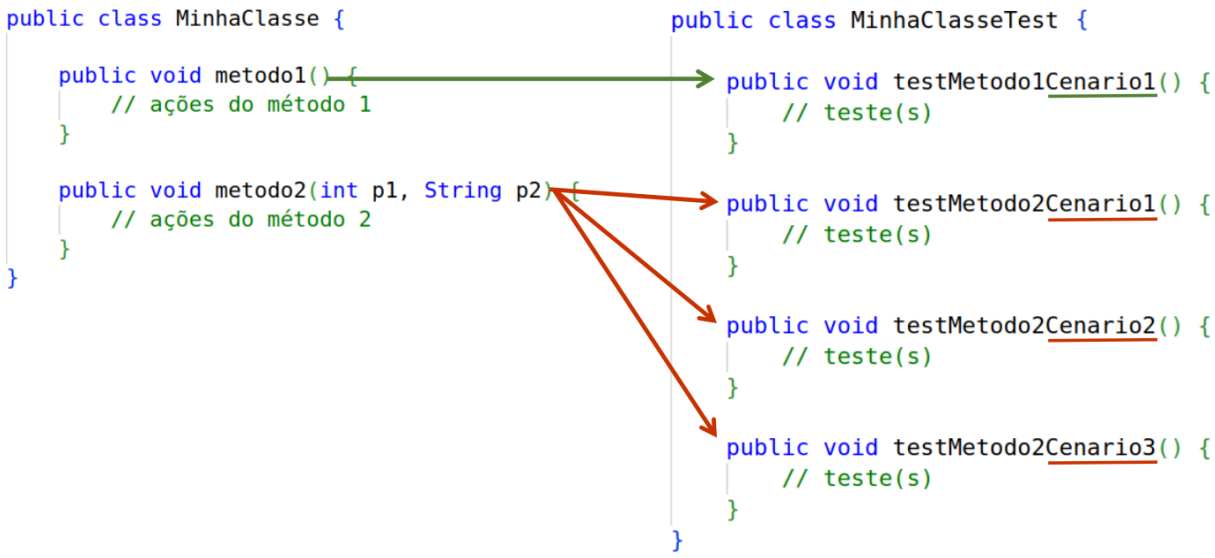


Figura 4.1: Classe a ser testada x classe de testes

A figura ilustra a seguinte situação: tínhamos que testar a `MinhaClasse`, que possui 2 métodos (`metodo1` e `metodo2`). Foi criada a classe de testes `MinhaClasseTest` com 4 métodos (cenários) de teste, sendo 1 apenas para testar o `metodo1` e 3 para testar o `metodo2`.

4.3 Fixture methods (métodos de montagem)

Imagine os testes de uma classe que acessa um banco de dados: antes da execução dos testes, seria bem útil configurar um banco de dados temporário, para evitar que a execução de testes crie "lixo" em uma base de dados real. Ao final da execução dos testes, esse banco de dados deveria ser completamente eliminado. Situações como a desse exemplo são comuns em testes unitários mais complexos e em testes de integração.

Nessas situações, nas quais é necessário preparar um ambiente para a execução dos testes e/ou executar ações após a execução, o uso de ***fixture methods*** (ou **métodos de montagem**, em tradução livre) é bastante recomendado. Eles são uma forma de implementar o ciclo de vida de testes proposto no padrão ***Four-phase test***.

Padrão *Four-phase test*

"Teste de quatro fases", em tradução livre. Foi proposto por Gerard Meszaros em seu livro clássico *XUnit Test Patterns – Refactoring Test Code*, de 2007. Esse padrão propõe que a execução dos cenários de testes possua quatro fases:

1. **Setup**, em que se configura o necessário para a execução e validação dos testes;
2. **Exercise**, em que é executada a ação que é alvo do teste;
3. **Verify**, em que é verificado se a ação fez o que era esperado;
4. **Teardown**, em que todos os recursos alocados pelo teste (objetos em memória, arquivos em disco, conexões com outros sistemas etc.) devem ser liberados.

O conceito de *Four-phase* pode ser encontrado em http://xunitpatterns.com/Four%20Phase%20Test.html e está ilustrado na figura a seguir:

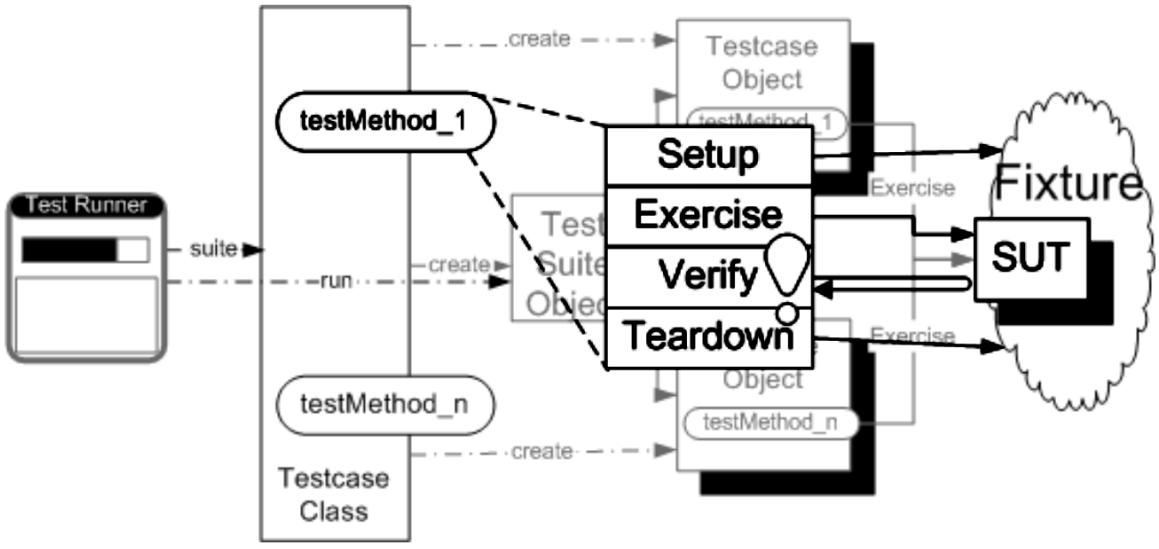


Figura 4.2: Padrão Four-phase test. Fonte: http://xunitpatterns.com/Four%20Phase%20Test.html

Os *fixture methods* do Spock framework são: `setup`, `setupSpec`, `cleanup` e `cleanupSpec`.

setup()

Caso exista na classe de testes, é executado **antes de cada método de teste**. Para deixar claro: caso existam 4 métodos de teste, esse método será executado 4 vezes, **antes** de cada um deles.

cleanup()

Caso exista na classe de testes, é executado **depois de cada método de teste**. Para deixar claro: caso existam 4 métodos de teste, esse método será executado 4 vezes, **depois** de cada um deles.

setupSpec()

Caso exista na classe de testes, é executado **apenas uma vez antes de todos os métodos de teste**. Para deixar claro: caso existam 4 métodos de teste, esse método será executado apenas 1 vez, **antes** da execução do primeiro teste. Caso exista também um `setup()` na mesma classe, sua primeira execução será **depois** do `setupSpec()`.

cleanupSpec()

Caso exista na classe de testes, é executado **apenas uma vez depois de todos os métodos de teste**. Para deixar claro: caso existam 4 métodos de teste, esse método será executado apenas 1 vez, **depois** da execução do último teste. Caso exista também um `cleanup()` na mesma classe, sua última execução será **antes** do `setupSpec()`.

A relação entre as fases do padrão *Four-phase test* e os *fixture methods* do Spock está representada na figura a seguir:

Fase	Método
Setup	setupSpec()
	setup()
Exercise	'método de teste'()
Verify	
Teardown	cleanup()
	cleanupSpec()

Figura 4.3: Relação entres as fases do Four-phase test e os fixture methods do Spock

Para deixar bem clara a ordem e quantidade de execuções dos *fixture methods*, observe a classe de testes de exemplo a seguir:

```
class MinhaClasseTest extends Specification {
  def setupSpec() {}

  def setup() {}

  def 'metodo1 deveria funcionar'() {}

  def 'metodo2 deveria funcionar'() {}

  def cleanup() {}

  def cleanupSpec() {}
}
```

Ao ser executada, os métodos seguiriam a ordem de execução indicada no diagrama de atividade a seguir:

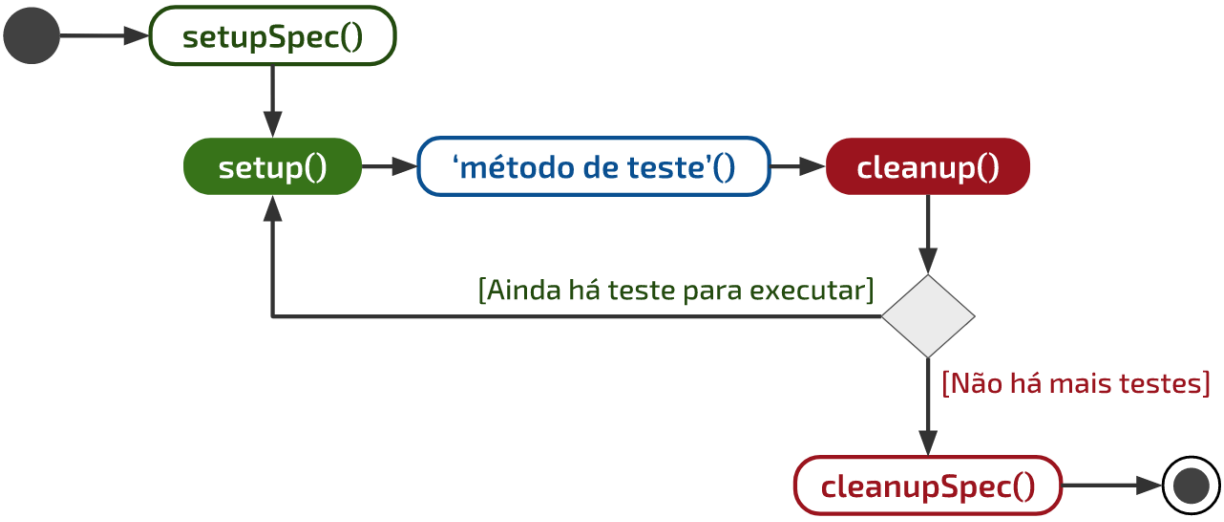


Figura 4.4: Ordem de execução dos fixture methods em uma classe de testes Spock

Importante: mesmo que os testes falhem por exceções ou por validações, todos os *fixture methods* são executados.

4.4 Blocks (blocos)

Imagine a situação em que uma classe de acesso a banco de dados possui vários métodos e cada um faz operações tão diferentes entre si que, para testá-los, seria necessário um banco de dados temporário diferente para cada. Ou seja, os diversos cenários de teste precisariam de *setups* e/ou de *cleanups* diferentes. Considerando apenas o recurso dos *fixture methods*, a saída seria criar várias classes de testes, uma para cada cenário de teste. Porém, para situações como essa, o Spock possui um interessante recurso chamado ***Blocks*** (**blocos**, em tradução livre), com o qual é possível **dividir as etapas de execução de um teste dentro de método de teste**.

Quanto à sua sintaxe, seu nome é **sempre** acompanhado por dois pontos (:) e cada bloco pode conter quantas linhas de código forem necessárias.

Os blocos do Spock indicam as principais etapas do ciclo de vida de um teste, desde a preparação até o pós-teste. São eles: `setup`, `given`, `when`, `expect`, `then`, `cleanup` e `and`.

Conforme a combinação de blocos utilizada, o teste acaba adotando o padrão ***Four-phase test*** ou o **Given-When-Then (GWT)**.

> **PADRÃO GIVEN-WHEN-THEN**
>
> "Dado-quando-então", em tradução livre. Foi proposto por Dan North e Ivan Moore no artigo *Introducing BDD*, publicado na revista *Better Software* em março de 2006. Esse padrão propõe que a execução dos cenários de testes possua 3 (três) fases:
>
> 1. **Given**, em que são definidas as precondições do cenário;
> 2. **When**, em que é identificado o evento (a ação) do teste;
> 3. **Then**, em que é verificado se o evento ocorreu conforme o esperado.

setup:

Caso seja necessária a configuração de um ambiente para a execução de um cenário de teste, ela pode ser feita no bloco `setup:`. Ele deve ser sempre o primeiro bloco e não pode ser usado em conjunto com o `given:`. Este bloco só pode estar presente 1 (uma) vez por método.

Se usado, torna-se necessária a presença de um bloco `expect:`.

given:

Caso seja necessária a configuração de um ambiente para a execução de um cenário de teste, ela pode ser feita no bloco `given:`. Ele deve ser sempre o primeiro bloco e não pode ser usado em

conjunto com o `setup:`. Este bloco só pode estar presente 1 (uma) vez por método.

Se usado, torna-se necessária a presença de um bloco `expect:` ou dos blocos `when:` e `then:`.

when:

É o bloco onde o evento a ser testado deve ser executado. Pode existir mesmo sem um bloco de preparação/configuração anterior. Se o bloco `given:` existir, deve estar necessariamente após ele. Não pode ser usado em conjunto com o `setup:`. Este bloco só pode estar presente 1 (uma) vez por método.

Se usado, torna-se necessária a presença de um bloco `then:`.

expect:

É o bloco onde o evento a ser testado deve ser executado. Pode ser usado se o bloco de preparação/configuração usado foi o `setup:` ou o `given:` e deve estar, necessariamente, após um deles. Este bloco só pode estar presente 1 (uma) vez por método.

É o **único** bloco que pode estar **sozinho** em um método de testes, como foi feito no capítulo anterior.

then:

É o bloco onde o evento a ser testado deve ser executado. Só pode ser usado se o bloco de preparação/configuração usado foi o `given:` e deve estar, necessariamente, após um `when:`. Este bloco só pode estar presente 1 (uma) vez por método.

cleanup:

Este bloco será executado após o bloco de execução do evento usado no teste (`expect:` ou `then:`) mesmo que eles falhem por verificação ou exceção. Deve estar, necessariamente, após o método de execução do evento. Este bloco só pode estar presente 1 (uma) vez por método.

and:

Este bloco simplesmente dá continuidade ao bloco imediatamente anterior, podendo ser usado após qualquer um dos outros blocos. É usado apenas para dividir e organizar um bloco em partes menores. Exemplos: se estiver após um `setup:` , funciona como um `setup:` ; se estiver após um `expect:` , funciona como um `expect:` . Este bloco pode estar presente várias vezes por método.

where:

Este bloco deve ser usado somente no caso de ***Data-Driven Testing***. Esse tipo especial de teste será abordado no capítulo 5.

Ordem e restrição de combinações de blocos

Os blocos não podem ficar em qualquer ordem e possuem algumas restrições quanto às combinações possíveis. Para ajudar a entender essas questões, veja a tabela a seguir:

Bloco	Ordem	Não pode ser usado em conjunto com
`setup:`	Sempre o 1º bloco	`given:` , `when:` e `then:`
`given:`	Sempre o 1º bloco	`setup:`
`when:`	1º bloco ou após `given:` (e seus `and:` , caso existam)	`setup:`

Bloco	Ordem	Não pode ser usado em conjunto com
`then:`	Sempre após `when:` (e seus `and:`, caso existam)	`setup:`
`expect:`	Sozinho no método ou após `setup:` (e seus `and:`, caso existam)	`given:` e `when:`
`cleanup:`	Sempre após `expect:` ou `then:` (e seus `and:`, caso existam)	–
`and:`	Após qualquer outro bloco	–

Blocos para Four-phase test

Caso você adote o padrão ***Four-phase test***, a combinação de blocos *recomendada* é `setup` > `and` > `expect`. A figura a seguir ilustra a relação entre as fases desse padrão e os blocos do Spock:

Fase	Bloco
Setup	setup:
Exercise	and:
Verify	expect:
Teardown	cleanup:

Figura 4.5: Relação entres as fases do Four-phase test e os blocos do Spock

Blocos para Given-When-Then

Caso você adote o padrão **Given-When-Then**, a combinação de blocos *recomendada* é `given` > `when` > `then`. A figura a seguir ilustra a relação entre as fases desse padrão e os blocos do Spock:

Fase	Bloco
Given	given:
When	when:
Then	then:

Figura 4.6: Relação entres as fases Given-When-Then e os blocos do Spock

4.5 Por que a anatomia do Spock é essa?

Talvez você tenha se perguntado: por que o Spock adota o termo *Specification* e não *Test* como o JUnit, por exemplo? Ou ainda: por que blocos explícitos para os padrões *Given-When-Then* e *Four-phase test*? É porque o Spock foi desenvolvido sob influência de uma técnica chamada **BDD**. Essa influência levou à anatomia dos testes do Spock abordada neste capítulo.

BEHAVIOR-DRIVEN DEVELOPMENT – BDD

"Desenvolvimento Dirigido a Comportamento", em tradução livre. Foi proposta por Dan North e Ivan Moore no artigo *Introducing BDD* (NORTH, 2006). Essa técnica fornece várias definições objetivas sobre como devem ser os testes em um projeto de desenvolvimento de software. Algumas dessas definições são:

- Nomes de métodos de teste devem ser **frases**, algo comum no Spock;
- Usar **linguagem ubíqua** (linguagem compreensível por programadores e não programadores, como analistas de negócio, por exemplo). O padrão *Given-When-Then* é um exemplo dessa linguagem e o Spock o implementa com os blocos de mesmo nome;
- Critérios de aceitação devem ser executáveis;
- "Comportamento" é uma palavra mais útil que "teste", e os "comportamentos" são criados em **especificações** (**Specifications**, por isso as classes de teste do Spock possuem esse sufixo, por padrão).

O artigo completo pode ser encontrado em https://dannorth.net/introducing-bdd/ (versão traduzida para português em https://www.dtsato.com/blog/work/introduzindo_desenvolvimento_orientado_comportamento_bdd/).

Como esta obra não tem o BDD como foco e tem como principal público-alvo profissionais sem experiência em testes, usaremos o termo **teste** em vez de **especificação**.

Spock é uma ferramenta para aplicação de BDD?

É possível usar o Spock na adoção BDD, como ensina e exemplifica Konstantinos Kapelonis em seu livro *Java Testing with Spock* (2015). Porém, na mesma obra, o autor afirma que o Spock não é uma ferramenta completa de BDD e indica outras mais adequadas para a adoção dessa técnica.

Conclusão

Neste capítulo, vimos os conceitos específicos do Spock e entendemos a anatomia de seus testes. Esses conceitos são a base para os demais capítulos. No próximo capítulo, veremos o fascinante recurso de ***Data-Driven Testing***, que facilita muito a criação de testes com múltiplos cenários.

CAPÍTULO 5
Data-Driven Testing – Facilitando os testes de múltiplos cenários

No capítulo anterior, estudamos e aplicamos os conceitos e convenções do framework Spock. Neste capítulo, veremos a funcionalidade chamada ***Data-Driven Testing*** (*Testes Orientados a Dados*, em tradução livre). Com ela, é possível testar vários cenários para uma mesma funcionalidade de uma forma muito simples.

5.1 Testando um analisador de IMC

Imagine que precisamos desenvolver um componente que informa a condição de saúde com base no *índice de massa corporal* (o famoso IMC). Esse componente recebe o IMC e o sexo de uma pessoa e devolve uma frase com a condição de saúde correspondente. Para facilitar nosso exemplo, não vamos usar a verdadeira tabela do IMC segundo a Organização Mundial de Saúde, mas uma versão resumida e com valores arredondados como a seguir:

Condição	IMC em mulheres	IMC em homens
Abaixo do peso	< 19	< 21
Peso normal	>= 19 .. < 26	>= 21 .. < 27
Acima do peso ideal	>= 26	>= 27

Para implementar esse componente, vamos criar um projeto Java e depois dois tipos ***enum*** em Java: `Sexo` e `FaixaImc`, descritos nos

códigos a seguir:

```
package br.com.livrospockframework.capitulo05.enums;

public enum Sexo {
    FEMININO, MASCULINO;
}
```

A *enum* `Sexo` serve para garantir que só sejam informados os valores que definimos previamente. Neste caso, `MASCULINO` e `FEMININO`.

```
package br.com.livrospockframework.capitulo05.enums;

public enum FaixaImc {

    ABAIXO("Abaixo do peso", 19, 21),
    NORMAL("Peso Normal", 26, 27),
    ACIMA("Acima do peso", 100, 100);
    // 100 é um IMC virtualmente impossível

    protected String descricao;
    protected double limiteMaximoFeminino;
    protected double limiteMaximoMasculino;

    // Construtor para os 3 atributos

    public static FaixaImc getFaixa(double imc, Sexo sexo) {
        FaixaImc retorno = values()[0];

        for (int i=1; i < values().length; i++) {

            FaixaImc atual = values()[i];
            FaixaImc anterior = values()[i-1];

            double limiteInferior = (sexo == Sexo.FEMININO) ?
anterior.limiteMaximoFeminino : anterior.limiteMaximoMasculino;
            double limiteSuperior = (sexo == Sexo.FEMININO) ?
atual.limiteMaximoFeminino : atual.limiteMaximoMasculino;
```

```
            if (imc >= limiteInferior && imc < limiteSuperior) {
                retorno = atual;
            }
        }

        return retorno;
    }

    // getDescricao()

}
```

A *enum* `FaixaImc` contém as condições de saúde das três faixas de nossa tabela de IMC. O método que precisaremos testar é o `getFaixa()`, que tem como argumentos o IMC e o sexo de uma pessoa. Ele procura e retorna a condição de saúde após análise das faixas de IMC e sexo. Em ambos os *enums*, o pacote é apenas uma sugestão.

5.2 Um método, vários cenários

Agora que temos o problema e uma proposta de implementação da solução, **o que precisamos testar?** A partir de **dois valores de entrada** (IMC e sexo), precisamos testar **um valor de saída** (condição de saúde). Só precisamos testar o método `getFaixa()` da *enum* `FaixaImc`.

O que será determinante para o sucesso de nossos testes será **a quantidade de cenários a serem testados**. Conseguiu contar quantos cenários devem ser testados para esse componente? São **três** condições possíveis para `MASCULINO` e **três** para `FEMININO`. Logo, temos que testar **seis cenários**, no mínimo.

Classe de testes para vários cenários SEM *Data-Driven Testing*

Vamos criar uma classe de testes, chamá-la de `FaixaImcTest` e criar os seis cenários mínimos usando os recursos que aprendemos até aqui sobre o Spock. Veja o código-fonte a seguir:

```
package livrospock.capitulo05;

class FaixaImcTest extends Specification {

    def 'IMC deve estar na faixa correta'() {
    expect:
    FaixaImc.getFaixa(18, Sexo.FEMININO) == FaixaImc.ABAIXO
    FaixaImc.getFaixa(21, Sexo.FEMININO) == FaixaImc.NORMAL
    FaixaImc.getFaixa(27, Sexo.FEMININO) == FaixaImc.ACIMA
    FaixaImc.getFaixa(20, Sexo.MASCULINO) == FaixaImc.ABAIXO
    FaixaImc.getFaixa(23, Sexo.MASCULINO) == FaixaImc.NORMAL
    FaixaImc.getFaixa(28, Sexo.MASCULINO) == FaixaImc.ACIMA
    }
}
```

Ao executar essa classe de teste, certamente os seis cenários de teste passarão. Porém temos muita **repetição de código**. Vamos tentar melhorar isso? E se reescrevêssemos o método de teste de nossa classe `FaixaImcTest` e ele ficasse como o próximo código?

```
def 'IMC deve estar na faixa correta'() {
    setup:
    def cenarios = [
    [imc:18, sexo: Sexo.FEMININO, condicao: FaixaImc.ABAIXO],
    [imc:21, sexo: Sexo.FEMININO, condicao: FaixaImc.NORMAL],
    [imc:27, sexo: Sexo.FEMININO, condicao: FaixaImc.ACIMA],
    [imc:20, sexo: Sexo.MASCULINO, condicao: FaixaImc.ABAIXO],
    [imc:23, sexo: Sexo.MASCULINO, condicao: FaixaImc.NORMAL],
    [imc:28, sexo: Sexo.MASCULINO, condicao: FaixaImc.ACIMA]
    ]

    expect:
    cenarios.each{
    assert FaixaImc.getFaixa(it.imc, it.sexo) == it.condicao
```

```
    }
}
```

O que achou? Note que organizamos melhor o teste, dividindo-o explicitamente nas fases `setup` (em que criamos uma tabela de cenários com as entradas e saídas em uma lista de mapas) e `expect` (em que iteramos na tabela de cenários e testamos cada item). Caso tenha ficado com dúvidas na lista, nos mapas ou na iteração usados no último código, não se esqueça de que pode recorrer ao *Apêndice A – Guia de Groovy para desenvolvedores Java*.

Classe de testes para vários cenários COM *Data-Driven Testing*

Percebeu que foi usado o termo "tabela de cenários" para descrever a lista de mapas com as entradas e saídas do teste anterior? Para poder descrever uma tabela de cenários literalmente como uma tabela, podemos usar o recurso de ***Data-Driven Testing*** do Spock.

Como o nome sugere, esse recurso pode ser usado quando temos situações parecidas com o exemplo do IMC, onde há vários valores de saída a partir de várias combinações de valores de entrada. Veja outros exemplos de componentes nos quais esse recurso ajudaria muito nos testes:

- Miniguia astral (entradas: dia e hora do nascimento / saídas: signo e ascendente);
- Gerador de contracheque (entradas: vencimento e benefícios / saídas: descontos legais e valor líquido);
- Simulador de resultados de um campeonato esportivo (entradas: resultados das partidas / saída: campeão).

Vejamos como ficaria nosso método de teste usando esse recurso no código a seguir:

```
def 'IMC deve estar na faixa correta'() {
    expect:
    FaixaImc.getFaixa(imc, sexo) == esperado
```

```
    where:
    imc | sexo           || esperado
    18  | Sexo.FEMININO  || FaixaImc.ABAIXO
    21  | Sexo.FEMININO  || FaixaImc.NORMAL
    27  | Sexo.FEMININO  || FaixaImc.ACIMA
    20  | Sexo.MASCULINO || FaixaImc.ABAIXO
    23  | Sexo.MASCULINO || FaixaImc.NORMAL
    28  | Sexo.MASCULINO || FaixaImc.ACIMA
}
```

Veja como a configuração da tabela de cenários ficou mais alto nível, sendo compreensível até por não programadores. Ela foi escrita, literalmente, em forma de tabela, o que reduziu a repetição e volume de código na classe de teste.

Quando usamos o recurso do *Data-Driven Testing*, o bloco `expect` vem sempre **antes** do bloco `where`. No `expect` indicamos o código a ser usado para o teste de cada cenário e no `where` configuramos quantos cenários acharmos necessários.

Um detalhe importante são os nomes das variáveis de entrada e saída: os `imc` e `sexo` usados pelo método `getFaixa()` e o `esperado` no bloco `expect` são os mesmos das colunas `imc`, `sexo` e `esperado` da tabela no bloco `where`. Essa relação está ilustrada na figura a seguir:

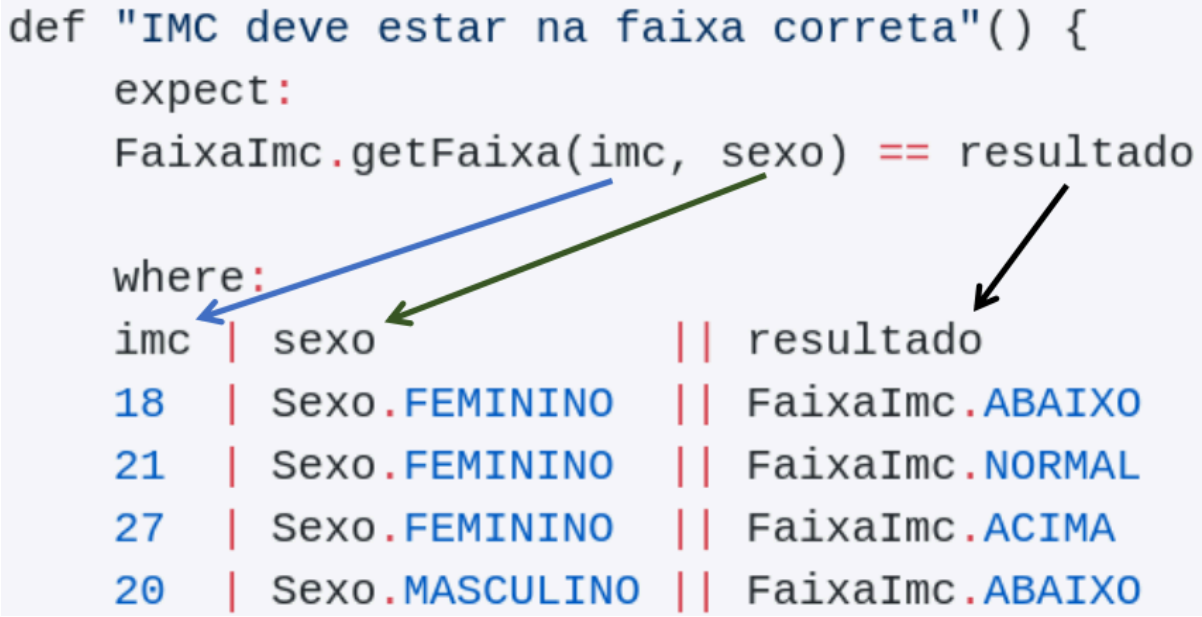


Figura 5.1: Entradas e saídas esperadas em um teste com Data-Driven Testing

Na tabela de cenários, usamos o separador ***pipe*** (`|`) como separador de coluna. Usamos dois *pipes* (`||`) antes da coluna de **esperado** para melhor organização, mas poderíamos ter usado apenas um também.

Certo, mas será que nossa *enum* `FaixaImc` já funciona perfeitamente? Afinal, nos testes anteriores, os cenários continham valores de IMC um tanto confortáveis, ou seja, que não ficavam nos limites entre as faixas. Vamos acrescentar cenários que testem os limites das faixas. Veja o próximo código:

```
def 'IMC deve estar na faixa correta'() {
    expect:
    FaixaImc.getFaixa(imc, sexo) == esperado

    where:
    imc    | sexo           || esperado
    18     | Sexo.FEMININO  || FaixaImc.ABAIXO
    21     | Sexo.FEMININO  || FaixaImc.NORMAL
    27     | Sexo.FEMININO  || FaixaImc.ACIMA
    20     | Sexo.MASCULINO || FaixaImc.ABAIXO
    23     | Sexo.MASCULINO || FaixaImc.NORMAL
```

```
    28     | Sexo.MASCULINO || FaixaImc.ACIMA
    18.99  | Sexo.FEMININO  || FaixaImc.ABAIXO
    19     | Sexo.FEMININO  || FaixaImc.NORMAL
    25.99  | Sexo.FEMININO  || FaixaImc.NORMAL
    26     | Sexo.FEMININO  || FaixaImc.ACIMA
    20.99  | Sexo.MASCULINO || FaixaImc.ABAIXO
    21     | Sexo.MASCULINO || FaixaImc.NORMAL
    26.99  | Sexo.MASCULINO || FaixaImc.NORMAL
    27     | Sexo.MASCULINO || FaixaImc.ACIMA
}
```

Veja como fica mais simples acrescentar novos cenários, apenas adicionando novas linhas à tabela do bloco `where`.

Testes com falha no *Data-Driven Testing*

Vamos supor que a `FaixaImc` tem um pequeno erro e falhe em alguns cenários. Nesse caso, o log de saída do teste terá a mesma estrutura que vimos nos capítulos anteriores, como o texto de exemplo a seguir:

```
FaixaImc.getFaixa(imc, sexo) == esperado
         |        |    |     |  |
         ABAIXO   19   |     |  NORMAL
                       |     false
                      FEMININO


    at livrospock.capitulo05.FaixaImcTest.imc deve estar na faixa correta
mesmo no limite(FaixaImcTest.groovy:52)
```

No log de exemplo, o teste falhou porque, para o cenário da 8ª linha da tabela de cenários, o resultado esperado para os IMC `19` e sexo `FEMININO` era `NORMAL`, porém o retorno do método `getFaixa()` foi `ABAIXO`.

Atenção! A falha na execução de um cenário não impede a execução dos seguintes. Por exemplo: se, dentre os 14 cenários, os das linhas 1 e 3 da tabela falharem, mesmo assim os cenários das linhas 2 e depois de 4 a 14 serão executados normalmente.

5.3 Descrevendo os testes que falham com detalhes dos cenários

No tópico anterior, vimos que os detalhes do erro em caso de falha no teste já são bem úteis. Porém, em situações com muitos cenários e/ou com um maior número de valores de entrada e saída, pode ser necessário detalhar mais ainda os erros. Para isso, existe o recurso de ***Unrolled method*** do Spock. A seguir, vejamos como transformar o exemplo anterior com esse recurso para nos dar mais detalhes quando um teste falhar:

```
import spock.lang.Unroll;

@Unroll
def "IMC #imc deve estar na faixa #esperado para o sexo #sexo"() {
    expect:
    FaixaImc.getFaixa(imc, sexo) == esperado

    where:
    imc    | sexo           || esperado
    18     | Sexo.FEMININO  || FaixaImc.ABAIXO
    // demais cenários
}
```

Usamos a anotação `@spock.lang.Unroll` sobre o método de teste para torná-lo um ***Unrolled method***. Assim, os trechos `#imc`, `#esperado` e `#sexo` do nome do método serão substituídos pelos valores das colunas de mesmo nome na tabela de cenários no log de saída em caso de falha de teste.

Um exemplo de log de saída em caso de falha para o cenário de IMC `19` e sexo `FEMININO` seria como o do texto a seguir:

```
IMC 19 deve estar na faixa NORMAL para o sexo FEMININO
(livrospock.capitulo05.FaixaImcTest)  Time elapsed: 0.007 sec  <<<
FAILURE!
org.spockframework.runtime.SpockComparisonFailure: Condition not
satisfied:
```

```
FaixaImc.getFaixa(imc, sexo) == esperado
         |        |    |     |  |
         ABAIXO   19   |     |  NORMAL
                       |     false
                      FEMININO
```

Note que, antes do texto mais técnico com os valores de objetos e métodos da linha de código, agora temos um texto contendo os valores explícitos do cenário que falhou.

O bloco `where` sempre fica após a execução do teste

O bloco `where` sempre fica **após** os blocos de teste, ou seja, sempre após `expect` ou `then`, dependendo da opção pelo ***Four-phase test*** ou **Given-When-Then (GWT)** (padrões abordados no capítulo 4, *Anatomia de um teste Spock*). Poderíamos, por exemplo, refazer o teste da tabela de IMC no padrão **GWT** conforme o código a seguir:

```
def "IMC deve estar na faixa correta"() {
    given:
    // código de "setup" (preparação para o teste)

    when:
    def resultado = FaixaImc.getFaixa(imc, sexo)

    then:
    resultado == esperado

    where:
    imc    | sexo           || esperado
    18     | Sexo.FEMININO  || FaixaImc.ABAIXO
    21     | Sexo.FEMININO  || FaixaImc.NORMAL
    27     | Sexo.FEMININO  || FaixaImc.ACIMA
}
```

No último exemplo, para aproveitar a mesma *enum* `FaixaImc` e usar o padrão **GWT**, criamos um objeto a partir de uma classe anônima com um método `getFaixa()` que apenas retorna o resultado do

método homônimo da *enum*. Quanto ao teste em si, a diferença foi a troca do `expect` por `when` + `then`. Note que aqui também o `where` é após o bloco de teste – que, no caso, é o `then`.

Conclusão

Neste capítulo, vimos o superútil e versátil recurso de *Data-Driven Testing* do Spock. Vimos como criar um teste de forma simples mesmo que seja executado para vários cenários. No próximo capítulo, veremos como **esperar que a execução de métodos lance exceções**.

Capítulo 6
Exception Conditions - Testes à espera de exceções

Até aqui já falamos sobre os fundamentos do Spock e sobre o recurso ***Data-Driven Testing***, que facilita a criação de testes para componentes que exigem que vários cenários sejam testados. Mas há situações, como as que veremos a seguir, nas quais é esperado que ocorram uma ou mais exceções. Neste capítulo, veremos como testar métodos quando esperamos que ocorram exceções com o recurso ***Exception Conditions***. Esse mecanismo do Spock permite testar se um componente lançou uma exceção, além de verificar os detalhes do erro.

6.1 Às vezes o certo é dar errado

Vamos continuar com o exemplo do capítulo anterior: o componente que dá informações baseadas no IMC. Vamos supor que precisamos agora calcular o IMC, algo que não fizemos anteriormente. O cálculo do IMC é simples. Veja a fórmula:

```
IMC = peso / altura x altura
```

A partir dessa fórmula, poderíamos criar uma calculadora de IMC com uma classe simples como a do código a seguir:

```
package br.com.livrospockframework.capitulo05;

public class CalculadoraImc {
    public double calcularImc(double peso, double altura) {
        return peso / altura * altura;
    }
}
```

Chamamos a classe de `CalculadoraImc`. Ela possui apenas um método, que é público, chamado `calcularImc` que, ao receber um `peso` e uma `altura`, retorna o IMC a partir da fórmula que adotamos.

Já vimos como testar o resultado do método nos capítulos anteriores. Mas, **e se algo der errado**? Aliás, reflita: *o que pode dar errado nesse cálculo?* Ou ainda: *esse cálculo deve ser efetuado para quaisquer valores de* `peso` *e* `altura`? Vamos supor que alguma pessoa desenvolvedora fez essas reflexões e chegou às seguintes conclusões sobre o método `calcularImc()`:

- Ele **não** pode receber `peso` com **valores menores ou iguais a 0 (zero)**. Afinal, seria um peso irreal;
- Ele **não** pode receber `altura` com **valores menores ou iguais a 0 (zero)**. Afinal, seria uma altura irreal.

Então, a pessoa desenvolvedora fez algumas mudanças no código do método `calcularImc()` para garantir que valores inválidos de `peso` e `altura` não provocariam a execução do cálculo e, consequentemente, a devolução de um IMC sem sentido. A nova versão do método está no código a seguir:

```
public double calcularImc(double peso, double altura) {
    if (peso <= 0) {
        throw new IllegalArgumentException("Peso inválido: "+peso);
    }
    if (altura <= 0) {
        throw new IllegalArgumentException("Altura inválida: "+altura);
    }
    return peso / altura * altura;
}
```

Em sua nova versão, o método `calcularImc()` não retorna o cálculo do IMC caso o `peso` ou `altura` recebidos sejam menores ou iguais a 0. Em vez disso, lança uma `IllegalArgumentException`, que é uma exceção da biblioteca padrão do Java.

Blocos `try-catch`

Caso você não se sinta seguro com seu conhecimento sobre tratamento de exceções e blocos `try-catch`, segue um resumo.

Quando podem ser usados?

Quando algum método anuncia uma **exceção checada** em sua assinatura. Isso obriga quem o invoca a tratar essa exceção, o que pode ser feito com esses blocos.

Quando algum método anuncia uma **exceção não checada** em sua assinatura, o que **não** obriga quem o invoca a tratá-la, o(a) desenvolvedor(a) pode usar esse bloco apenas se achar necessário. Isso costuma ser feito em situações como: registrar a ocorrência da exceção em um log de erro; evitar que um bloco de linhas código seja interrompido bruscamente; envelopar uma exceção em outra, normalmente por uma criada no projeto.

Como funcionam?

Dentro do bloco `try` há uma ou mais linhas que podem lançar exceções. Se nenhuma exceção ocorrer, todas as linhas desse bloco serão executadas.

Caso alguma exceção ocorra no bloco `try`, as linhas de código do bloco `catch` serão executadas. Na declaração do `catch` devem constar uma ou mais classes de exceções e o objeto no qual é instanciada a exceção. A esse objeto chamamos de **exceção capturada**. Vejamos o código a seguir:

```
try {
  // linha a
  // linha b
} catch (IOException ex) {
  // linha c
}
```

No código de exemplo, se nenhuma exceção ocorrer no bloco `try`, as *linhas a* e *b* serão executadas. Caso contrário, a *linha c* será executada, pois está no bloco `catch`.

O bloco `catch`, se executado, possui à sua disposição na *linha c* o objeto `ex` (do tipo `IOException`), que é a exceção capturada. Caso fosse necessário tratar mais de uma exceção ao mesmo tempo, poderíamos anunciar mais de um tipo de exceção no bloco `catch`, como no próximo exemplo:

```
} catch (IOException | NumberFormatException ex) {
  // linha c
}
```

Outra opção é ter mais de um bloco `catch`, como no exemplo:

```
} catch (IOException ex1) {
  // linha c
} catch (NumberFormatException ex2) {
  // linha d
}
```

Nesse último exemplo, caso a exceção que ocorra no bloco `try` seja uma `IOException`, a *linha c* seria executada. Caso ocorra uma `NumberFormatException`, a *linha d* seria executada.

Como foi dito, isso é apenas um resumo. Para saber mais sobre os blocos `try-catch` e tudo mais que envolve exceções com Java, uma ótima referência é o livro da Casa do Código *Java SE 8 Programmer I - O guia para sua certificação Oracle Certified Associate*, de Guilherme Silveira e Mário Amaral.

Para garantir que a execução do método `calcularImc()` lance uma exceção, vamos criar uma classe de teste chamada `CalculadoraImcTest` e usar os recursos que aprendemos até o momento para tentar testar a situação de erro desse método em caso de `peso` inválido.

```
package livrospock.capitulo05;

import br.com.livrospockframework.capitulo05.CalculadoraImc

class CalculadoraImcTest extends Specification {

    def 'lançar exceção com peso inválido'() {
        when:
        boolean houveExcecao
        try {
            new CalculadoraImc().calcularImc(0, 1.70)
            houveExcecao = false
        } catch (IllegalArgumentException e) {
            houveExcecao = true
        }

        then:
        houveExcecao
    }
}
```

O teste recém-criado passará caso a execução do `calcularImc()` lance uma exceção do tipo `IllegalArgumentException`. Porém ficou um código um tanto verboso: tivemos que usar blocos `try-catch` para a montagem do teste. Caso o bloco `try` chegue ao seu final sem exceções, a variável `houveExcecao` recebe `false`. Caso uma `IllegalArgumentException` ocorra, `houveExcecao` recebe `true`. O valor de `houveExcecao` é que determina o resultado do teste.

É em situações como essa, em que precisamos testar se houve erro, que podemos usar o recurso *Exception Conditions* do Spock. Veja como faríamos o mesmo teste usando esse recurso no código a seguir:

```
def 'lançar exceção com peso inválido'() {
    when:
    new CalculadoraImc().calcularImc(0, 1.70)

    then:
```

```
    thrown(IllegalArgumentException)
}
```

Aqui usamos o método `thrown()` da superclasse `Specification`. Esse método é uma das formas de usar *Exception Conditions*. Ele verifica se a exceção do argumento foi lançada durante a execução do teste. Ao ser executado, esse teste passaria normalmente.

Mas vamos supor que ele tivesse sido executado quando a classe `CalculadoraImc` ainda estava na primeira versão apresentada neste capítulo. Como não havia a possibilidade de ocorrer uma exceção, esse teste falharia e geraria um log como o do texto a seguir:

```
lançar exceção com peso inválido
(livrospock.capitulo05.CalculadoraImcTest)  Time elapsed: 0.001 sec  <<<
FAILURE!

org.spockframework.runtime.WrongExceptionThrownError: Expected exception
of type 'java.lang.IllegalArgumentException', but no exception was thrown
```

Note no log de exemplo que o erro foi que não aconteceu o erro esperado na execução do `calcularImc()`. Não foi lançada uma `IllegalArgumentException`.

Um detalhe importante: o método `thrown` **só pode estar dentro de blocos** `then`. Se estiver dentro de um `expect`, a classe de testes sequer compila.

6.2 Testando vários cenários de erro

Um componente pode ter vários cenários que provoquem erro. E é o caso de nosso método `calcularImc()`, que pode lançar exceções para valores de `peso` ou `altura` inválidos. Para testar os vários cenários, poderíamos simplesmente criar vários métodos de teste, um para cada cenário. Porém, isso é desnecessário com o Spock, já que é possível repetir os blocos `when` e `then` para testar diferentes

cenários que esperam exceções. Vejamos como testar mais de um cenário que provoque exceção para valores inválidos de `peso` no próximo código:

```
def 'lançar exceção com peso inválido'() {
    when:
    new CalculadoraImc().calcularImc(0, 1.70)

    then:
    thrown(IllegalArgumentException)

    when:
    new CalculadoraImc().calcularImc(-1, 1.70)

    then:
    thrown(IllegalArgumentException)
}
```

6.3 Verificando detalhes das exceções

Até agora vimos como verificar se uma determinada exceção foi lançada com a execução de um método. Mas pode ser necessário testar se a exceção que ocorreu possui detalhes desejados.

Na última versão do método de cálculo de IMC apresentada, as exceções continham mensagens específicas para erros de `peso` e `altura`. Vamos supor que seja necessário garantir que as mensagens sejam exatamente aquelas. Poderíamos criar um novo método de teste na `CalculadoraImcTest`, como o do código:

```
def 'lançar exceção c/ mensagem correta p/ peso inválido'() {
    when:
    def peso = -1
    new CalculadoraImc().calcularImc(peso, 1.70)

    then:
```

```
    def ex = thrown(IllegalArgumentException)
    ex.message == "O peso $peso é inválido"
}
```

Note que a diferença para o que fizemos até agora é que usamos o retorno do método `thrown()` para guardar a exceção lançada no objeto `ex`. Com o `ex` à disposição, podemos testar quaisquer detalhes que precisarmos da exceção. No exemplo, verificamos se a mensagem da exceção contém um determinado texto.

De propósito, testamos uma mensagem diferente da que realmente é usada na exceção lançada no método de cálculo do IMC. Ao executar o teste, ele falhará e produzirá um log de erro, como o do próximo texto:

```
lançar exceção c/ mensagem correta p/ peso
inválido(livrospock.capitulo05.CalculadoraImcTest)  Time elapsed: 0.015
sec  <<< FAILURE!
org.spockframework.runtime.SpockComparisonFailure: Condition not
satisfied:

ex.message == "O peso $peso é inválido"
|  |       |           |
|  |       |           -1
|  |       false
|  |       14 differences (30% similarity)
|  |       (P--)eso (-----)inválido(: -1.0)
|  |       (O p)eso (-1 é )inválido(------)
|  Peso inválido: -1.0
java.lang.IllegalArgumentException: Peso inválido: -1.0
```

Para que o último teste passe, basta ajustar o bloco `then` para esperar pela mensagem correta, como no próximo código:

```
then:
def ex = thrown(IllegalArgumentException)
ex.message == "Peso inválido: ${peso.toDouble()}"
```

Nos exemplos deste capítulo verificamos o conteúdo da mensagem da exceção lançada, mas poderíamos ter testado qualquer outro

atributo ou retorno de método de `ex` que fosse necessário. Por exemplo, poderíamos verificar a exceção anterior na pilha de exceções, por meio do método `getCause()`.

Conclusão

Neste capítulo, vimos como testar situações de exceções esperadas com o uso de *Exception Conditions* do Spock. Vimos como criar um teste que verifica se determinada exceção foi lançada, bem como testar seus detalhes. No próximo capítulo, veremos como criar e configurar **Mocks**.

CAPÍTULO 7
Testando com a ajuda de mocks

Até aqui já vimos como criar vários tipos de testes com Spock: testes simples, testes que exigem vários cenários e testes que esperam por exceções. Neste capítulo, veremos como trabalhar com *dublês*, que são objetos que imitam o comportamento de componentes que dão suporte a um teste, mas que não são seus protagonistas.

7.1 O que são dublês e por que usá-los

Para a execução de alguns tipos de testes, fica complexo e até arriscado usar todos os componentes reais envolvidos na funcionalidade a ser testada. Imagine que você precisa testar funcionalidades como: um método que exclui centenas de registros em um banco de dados; um método que envia centenas de e-mails; um método que traz centenas de objetos JSONs de uma REST API. Em todos esses exemplos estamos sujeitos a uma série de problemas.

Ou seja, há testes nos quais usar todos os componentes reais envolvidos pode torná-los mais **complexos** do que deveriam. Outro problema no uso de certos tipos de componentes é o fato de estarem expostos à **dependência de fatores externos**. Existem ainda componentes cujo uso em testes pode ser até **perigoso**. São exemplos de componentes com uma ou mais dessas características aqueles que:

- Exigem muitas configurações;
- Exigem muitas dependências;
- Têm necessidade de estar em uma rede;

- Têm necessidade de acesso à internet;
- São afetados pela latência de rede;
- Têm a necessidade de licença de uso de software;
- Dependem da disponibilidade de algum serviço externo;
- Dependem do desempenho de algum serviço externo;
- Podem afetar dados sensíveis em bases de dados;
- Podem realizar ações que geram inconvenientes, como o envio excessivo de e-mails para clientes reais, por exemplo;
- Podem levar ao aumento de custos financeiros, pois podem consumir serviços que são tarifados por tempo e/ou volume de uso.

Porém, imagine que, nos exemplos citados no início deste tópico:

- Os registros não precisam ser realmente excluídos do banco. Só é necessário garantir que houve a solicitação para excluí-los;
- Os e-mails não precisam ser realmente enviados. Só é necessário garantir que ocorreram as solicitações de envio;
- Não é necessário realmente obter os JSONs da API. Só é necessário ter objetos JSONs na quantidade e no formato desejados.

Note que as funcionalidades que poderiam acrescentar complexidade e/ou perigo aos testes **não são o alvo dos testes**. Para situações assim, podemos substituir alguns componentes por **dublês**, que são objetos que imitam outros.

Existem vários tipos de dublês, sendo os principais *dummy*, *fake*, *stub*, *spy* e *mock*. Esses termos foram sugeridos por Gerard Meszaros em *xUnit Test Patterns: Refactoring Test Code*, de 2007 (http://xunitpatterns.com/Mocks,%20Fakes,%20Stubs%20and%20Dummies.html), e também usados por Martin Fowler em um artigo seu no mesmo ano (https://martinfowler.com/articles/mocksArentStubs.html).

Tipos de dublês

Dummy

É o tipo de dublê mais simples. É usado apenas para preencher passagens de parâmetros. São valores simples como pequenos alfanuméricos, números, uma data qualquer, textos em branco ou até valores nulos.

Fake

Esse tipo de dublê é usado principalmente para contornar problemas de dependências. São objetos simples, que não guardam estado. Um exemplo seria criar uma classe `FakeDriver` que imita uma ***Driver*** (um Driver JDBC), possuindo métodos públicos com a mesma assinatura. Assim, dispensamos toda uma lista de dependências e podemos realizar o teste cujo foco não era o Driver JDBC, e sim o que acontece após invocar alguma funcionalidade dele.

Os *fakes* sempre apresentam o mesmo comportamento para suas funcionalidades. Por exemplo, um objeto *fake* de uma classe `FakeDriver` sempre retornaria o mesmo valor em um método `connect()` independentemente dos argumentos usados. Devido à sua simplicidade, são normalmente criados manualmente, ou seja, sem auxílio de bibliotecas de *mocks*.

Stub

É muito parecido com o *fake*. A diferença é que podem ser configurados diferentes comportamentos para os métodos conforme os argumentos usados. Ou seja, um mesmo objeto *stub* pode ter diferentes comportamentos para diferentes cenários de teste. São normalmente criados com auxílio de alguma biblioteca de *mocks*.

Spy

São dublês que permitem saber detalhes que ocorreram durante a execução de seus métodos. Por exemplo, um Driver JDBC deveria receber 3 vezes a solicitação para invocar seu método para executar instruções SQL caso o método que está sendo testado receba uma

lista com 6 elementos. Se usarmos um *spy* para imitar o Driver JDBC, podemos obter dele a informação de quantas vezes qualquer um de seus métodos foi executado durante um teste. São normalmente criados com auxílio de alguma biblioteca de *mocks*.

Mocks

São os dublês mais completos. Permitem a configuração de diferentes comportamentos de acordo com as entradas (como um *stub*), assim como permitem saber quantas vezes seus métodos foram executados (como um *spy*). São normalmente criados com auxílio de alguma biblioteca de *mocks*.

Dublês dos tipos *dummy* e *fake* são criados manualmente, sem auxílio do framework, e, por esse motivo, não são foco deste capítulo.

7.2 Dependências adicionais para mocks

Para que seja possível trabalhar com *mocks* no Spock, é necessário incluir a dependência de duas bibliotecas: `objenesis` e `byte-buddy`. Elas serão usadas de forma transparente pelo Spock. A seguir, o código que deve ser incluído no `pom.xml` do projeto para adicioná-las:

```
<dependency>
    <groupId>net.bytebuddy</groupId>
    <artifactId>byte-buddy</artifactId>
    <version>1.12.23</version> <!-- ou a versão que precisar -->
    <scope>test</scope>
</dependency>
<dependency>
    <groupId>org.objenesis</groupId>
    <artifactId>objenesis</artifactId>
    <version>3.3</version> <!-- ou a versão que precisar -->
```

```
    <scope>test</scope>
</dependency>
```

A versões dessas dependências podem ser as de que precisar ou preferir. A sugestão é deixá-las no escopo `test` do projeto, para evitar seu uso por engano fora dos testes.

Importante! *Caso esteja usando Spring, essas duas dependências não são necessárias.*

7.3 Configurando retornos de métodos em mocks

O tipo de *mock* mais simples de que podemos precisar é aquele em que apenas programamos os retornos de seus métodos. Imagine uma situação em que precisamos invocar métodos de um objeto cuja criação é muito difícil de reproduzir usando sua classe real (é instanciada via inversão de controle, por exemplo), porém o que realmente precisamos para um teste é apenas usar o retorno de alguns de seus métodos. Para situações como essa, é possível criar e configurar o retorno de métodos de *mocks*.

Vamos usar como exemplo a mesma *Calculadora de IMC* dos capítulos anteriores. Vamos supor que foi necessário evoluir o projeto e foi criada uma interface chamada `PedidoImc` cuja implementação em tempo de execução ocorre por meio de mecanismos complexos de reproduzir, como inversão de controle, por exemplo. Seu código-fonte está a seguir:

```
package br.com.livrospockframework.capitulo07.model;

import br.com.livrospockframework.capitulo05.enums.Sexo;

public interface PedidoImc {
    String getNome();
```

```
    double getPeso();
    double getAltura();
    Sexo getSexo();
}
```

E para representar o resultado de um cálculo de IMC, foi criada também uma classe, a `ResultadoImc`, cujo código está a seguir:

```
package br.com.livrospockframework.capitulo07.model;

public class ResultadoImc {
    private String nome;
    private double imc;
    private String condicao;

    public ResultadoImc(String nome, double imc) {
        super();
        this.nome = nome;
        this.imc = imc;
    }

    // construtor padrão, getters e setters
}
```

O atributo `nome` é o nome da pessoa de quem foi solicitado o cálculo. O `imc` é o valor do IMC calculado e `condicao` é a descrição da condição de saúde da pessoa correspondente ao seu IMC.

A interface e a classe recém-criadas agora são usadas na classe `CalculadoraImc`, a mesma dos capítulos anteriores. Observe o método `verificarCondicaoImc()` criado nesta classe:

```
import br.com.livrospockframework.capitulo07.model.PedidoImc;
import br.com.livrospockframework.capitulo07.model.ResultadoImc;

// demais imports

public class CalculadoraImc {
    // mesmo método calcularImc() dos capítulos anteriores
```

```
    public ResultadoImc verificarCondicaoImc(PedidoImc pedido) {
        ResultadoImc resultado = new ResultadoImc();
        resultado.setNome(pedido.getNome());

        try {
            double imc = this.calcularImc(pedido.getPeso(),
pedido.getAltura());
            resultado.setImc(imc);
            resultado.setCondicao(
                    FaixaImc.getFaixa(imc,
pedido.getSexo()).getDescricao());
        } catch (IllegalArgumentException e) {
            resultado.setCondicao("Impossível calcular IMC:
"+e.getMessage());
        }

        return resultado;
    }
}
```

O método `verificarCondicaoImc()` recebe um objeto do tipo `PedidoImc` e retorna um objeto do tipo `ResultadoImc`. Esse método preenche o atributo `nome` do resultado com o valor do atributo de mesmo nome do pedido. Caso o IMC seja calculado, seu valor é atribuído ao atributo `imc` do resultado e o atributo `condicao` é preenchido conforme a descrição obtida em `FaixaImc`. Caso o IMC não seja calculado devido a valores inválidos de peso ou altura, a `condicao` recebe um texto padrão sobre a impossibilidade do cálculo.

Vamos então criar nosso primeiro *mock* com Spock para podermos testar o método `calcularImc()` da `CalculadoraImc`. Ele será criado na classe de teste já existente `CalculadoraImcTest`, como pode ser visto no método do código-fonte a seguir:

```
import br.com.livrospockframework.capitulo07.model.PedidoImc
import br.com.livrospockframework.capitulo07.model.ResultadoImc
// outros imports

class CalculadoraImcTest extends Specification {
```

```
    def 'deve ser um homem acima do peso'() {
        given:
        def pedido = Mock(PedidoImc)

        pedido.getNome() >> "Zé Fofinho"
        pedido.getSexo() >> Sexo.MASCULINO
        pedido.getPeso() >> 72
        pedido.getAltura() >> 1.50

        when:
        def resultado = new CalculadoraImc().verificarCondicaoImc(pedido)

        then:
        resultado.imc == 32
        resultado.condicao == FaixaImc.ACIMA.descricao
    }

    // demais testes
}
```

A criação do *mock* ocorreu no bloco `given`, onde usamos o método `Mock()` que vem da superclasse `spock.lang.Specification` do Spock. Esse método exige como argumento um tipo que indica para qual classe será criado um dublê. Logo, no código do exemplo, criamos um *mock* para a interface `PedidoImc`, que chamamos de `pedido`.

Logo após a criação do *mock*, programamos os retornos dos métodos de que precisaremos em nosso teste. Por meio do operador `>>` configuramos os retornos dos métodos `getNome()`, `getSexo()`, `getPeso()` e `getAltura()`. Se não tivéssemos configurado explicitamente os retornos desses quatro métodos, o `pedido` retornaria valores padrões de acordo com os tipos de retorno de cada método.

O QUE MÉTODOS DE *MOCKS* RETORNAM?

Quando acabamos de criar um *mock*, podemos invocar todos os métodos da classe original. Mas, se não programarmos seus retornos, o que eles retornam? Eles retornam valores padrão conforme seus tipos.

Tipos primitivos: métodos com retornos de tipos primitivos retornam o valor padrão de cada tipo primitivo conforme define a linguagem Java:

- Para `int`, `short` e `byte`, o valor padrão é `0`;
- Para `long`, é `0L`;
- Para `double`, é `0.0`;
- Para `float`, é `0.0f`;
- Para `char`, é `'\u0000'`;
- Para `boolean`, é `false`.

Classes (inclusive *Wrappers*): métodos com retorno de classes retornam `null` por padrão. Métodos com retorno de classes *wrappers* (`Integer`, `Double`, `Boolean` etc.) também retornam `null` por padrão.

Vetores e coleções: métodos com retorno de vetores e coleções retornam `null` por padrão, mesmo que se trate de um vetor de um tipo primitivo.

Métodos sem retorno (*void*): por padrão, métodos sem retorno (retorno `void`) são executados sem erro. A execução é instantânea, como se o método nem tivesse sido executado.

Assim, com o *mock* criado e configurado, durante a invocação do método `verificarCondicaoImc()`, o argumento do tipo `PedidoImc` retorna os valores programados no *mock* do teste. Ao final da execução, esse método retorna uma instância de `ResultadoImc` que pode ser analisada normalmente no bloco `then` do teste.

Só é possível indicar um retorno para métodos dos *mocks*? E se precisar de algo mais dinâmico?

Na verdade, o operador `>>` pode ser usado também para indicar **todo o corpo do método de um *mock***. No código do exemplo anterior, poderíamos, por exemplo, ter definido o retorno do método que indica o nome a partir do retorno do método que indica o sexo, como no código a seguir:

```
def 'deve ser um homem acima do peso'() {
    given:
    def pedido = Mock(PedidoImc)

    pedido.getNome() >> {
        if (this.getSexo() == Sexo.Masculino) {
            "Zé Fofinho"
        } else {
            "Maria Fofinha"
        }
    }

    // demais código do teste
```

Nesse código, indicamos não apenas um valor de retorno do método `getNome()`, mas todo o seu corpo, no qual definimos o retorno conforme o que retornar o método `getSexo()` do próprio *mock*.

Configurando métodos de *mocks* para aceitarem qualquer valor

Imagine a situação em que você precisa configurar o retorno de um método que possui um ou mais argumentos. Só que o valor de algum dos argumentos (ou todos) não é relevante para o teste ou é impossível de prever, ou podem receber qualquer valor. Nesse caso, podemos configurar o Spock para determinar o retorno do método com um ou mais argumento recebendo qualquer valor válido.

Vamos supor que a interface `PedidoImc` possua mais dois métodos com argumentos, como no código a seguir:

```
public interface PedidoImc {
    String getNome(String pronomeTratamento);
    String getNome(String pronomeTratamento, String apelido);
    // demais métodos
}
```

Para configurar o retorno das duas novas versões do método `getNome()` sem nos importarmos com os valores dos argumentos, poderíamos configurar um *mock* de `PedidoImc` usando o operador ***underline*** (_), como no código de teste Spock a seguir:

```
def pedido = Mock(PedidoImc)

pedido.getNome(_) >> "Sr. Zé Importante"
pedido.getNome(_, _) >> "Sr. Zé Imitador, vulgo 'ator'"
pedido.getNome(_, 'alpinista') >> "Sr. Zé Luiz, vulgo 'alpinista'"
pedido.getNome('Sr.', _) >> "Sr. Zé Boleiro, vulgo 'messi'"
```

Nesse código realizamos quatro configurações de retornos para as duas novas versões do método `getNome()`. Foram todas aplicadas no mesmo objeto *mock*. Os retornos ficaram assim configurados:

- **Na 1ª configuração**: ao ser invocado com qualquer valor no único argumento, retorna `"Sr. Zé Importante"`;
- **Na 2ª configuração**: ao ser invocado com quaisquer valores no primeiro e segundo argumentos, retorna `"Sr. Zé Imitador, vulgo 'ator'"`;
- **Na 3ª configuração**: ao ser invocado com qualquer valor no primeiro argumento e `'alpinista'` no segundo, retorna `"Sr. Zé Luiz, vulgo 'alpinista'"`;
- **Na 4ª configuração**: ao ser invocado com `'Sr.'` no primeiro argumento e qualquer valor no segundo argumento, retorna `"Sr. Zé Boleiro, vulgo 'messi'"`.

7.4 Configurando mocks para lançarem exceções

Imagine situações em que o comportamento de um componente pode lançar exceções em situações complexas de reproduzir a qualquer momento, como problemas de rede ou indisponibilidade de serviço externo, por exemplo. Para situações como essas, é possível configurar um *mock* para lançar exceções na invocação de métodos.

Criando um *mock* que lança uma exceção simples

Vamos supor que você precisa de um componente de envio de e-mails. Para facilitar esse tipo de tarefa, você optou por usar a biblioteca *Apache Commons Email* (https://commons.apache.org/proper/commons-email/) e criou a classe `EnvioEmailService`, cujo código-fonte está a seguir:

```
package br.com.livrospockframework.capitulo07.services;

import org.apache.commons.mail.Email;
import org.apache.commons.mail.EmailException;

public class EnvioEmailService {

    private Email email;

    private int enviados;

    public EnvioEmailService(Email email) {
        this.email = email;
    }

    public void enviarEmails(String assunto, String conteudo, List<String>
destinatarios) throws EmailException {
        this.email.setSubject(assunto);
        this.email.setMsg(conteudo);
        this.email.addTo(destinatarios.toArray(new
String[destinatarios.size()]));
```

```
            this.email.send();

            this.enviados++;
        }

        // getEnviados()
}
```

A classe `EnvioEmailService` possui um atributo do tipo `org.apache.commons.mail.Email`, que é atribuído no construtor da classe. É em objetos dessa classe da *Apache Commons Email* que são configurados os dados de acesso ao servidor de e-mail.

O método `enviarEmails()` usa seus argumentos para invocar os métodos do atributo de instância `email` e, caso tudo dê certo, o atributo `enviados` aumenta seu valor em **1**. Dentre os métodos invocados, o método `send()` anuncia uma `org.apache.commons.mail.EmailException`. Ao estudar a API da *Apache Commons Email*, verificamos que essa exceção sempre traz o motivo do erro. Exemplos:

- Para problemas de autenticação, a causa é uma `javax.mail.AuthenticationFailedException`;
- Para problemas de conexão, a causa é uma `com.sun.mail.util.MailConnectException`.

Existem várias outras causas de erro para o envio de e-mails, mas vamos usar apenas essas duas em nosso exemplo. Caso o e-mail seja enviado em erros, o atributo de instância `enviados` aumenta seu valor em 1.

Se não fosse possível criar *mocks* para configurar o lançamento das exceções no envio de e-mail, teríamos que ficar solicitando de fato o envio de e-mails. Os possíveis problemas dessa abordagem seriam vários: estaríamos sujeitos à latência de rede; poderíamos bloquear uma conta ou mesmo cair em uma *blacklist* ao tentar muitas autenticações sem sucesso; seria complicado simular problemas de

conexão de rede durante os testes; poderíamos acabar tendo que pagar pelo excessivo envio de e-mails, dependendo do contrato com o provedor.

Vamos então criar uma classe de teste chamada `EnvioEmailServiceTest`, na qual vamos usar *mocks* que lançam `org.apache.commons.mail.EmailException` com a devida causa para evitar que, para testar nossa `EnvioEmailService`, precisemos enviar e-mails de fato. Vejamos o código dessa classe de teste a seguir:

```
package livrospock.capitulo07

import org.apache.commons.mail.Email
import br.com.livrospockframework.capitulo07.services.EnvioEmailService
// demais imports

class EnvioEmailServiceTest extends Specification {

    Email email
    EnvioEmailService service

    def setup() {
        this.email = Mock(Email)
        this.service = new EnvioEmailService(this.email)
    }
}
```

A parte do código da `EnvioEmailServiceTest` por enquanto não possui testes. Nela, declaramos dois atributos: o `email`, do tipo `org.apache.commons.mail.Email`, que será instanciado com um *mock*, e o `service` do tipo da classe que queremos testar, a `EnvioEmailService`. O *mock* é criado no método `setup()`, onde é usado como argumento do construtor usado para instanciar o `service`. Vale lembrar que o `setup()` será invocado sempre antes de cada método de teste.

Agora vamos criar um teste que espera por exceção ocasionada por problema de conexão com o servidor de e-mail. Para isso, vamos

configurar o *mock* para lançar a exceção esperada para que ela ocorra quando usamos o `service`. Vejamos no próximo código:

```
// pacote

import javax.mail.AuthenticationFailedException
// demais imports

class EnvioEmailServiceTest extends Specification {

    // atributos e setup()

    def 'deve lançar erro de autenticação'() {
        given:
        def msgErro = 'Falha de autenticação! Sorry :('

        this.email.send() >> {
            throw new EmailException(new
AuthenticationFailedException(msgErro))
        }

        when:
        this.service.enviarEmails('assunto', 'conteudo', ['em@t.com'])

        then:
        def ex = thrown(EmailException)
        ex.cause.class == AuthenticationFailedException.class
        ex.cause.message == msgErro

        !this.service.enviados
    }
}
```

No bloco `given` apenas definimos a mensagem de erro que será usada no *mock* e para posterior comparação com o resultado da execução do teste. Em seguida, configuramos nosso *mock*, o `email`, para que, quando invocar o método `send()`, lance uma `EmailException` cuja causa é uma `javax.mail.AuthenticationFailedException`. Observe que definimos um

bloco de código com o operador `>>`. É que, com Spock, podemos definir todo o corpo do método do *mock*, e foi isso que foi feito no exemplo.

No bloco `when` apenas invocamos o método que queremos testar. E no bloco `then` testamos:

- Se foi lançada uma `EmailException`;
- Se a causa da exceção foi uma `AuthenticationFailedException`;
- Se a mensagem da causa foi a definida no *mock* (lá no bloco `given`);
- Se o atributo `enviados` continua zerado.

Ratificando: sem o uso de *mock*, nosso teste seria mais lento na execução (pois estaríamos sujeitos à latência de rede e desempenho do servidor de e-mail), além de correr o risco de bloquear uma conta devido às falhas de autenticação ou mesmo de entrar em uma *blacklist* do serviço de e-mail. E, com o uso de *mock*, não precisamos configurar a autenticação do objeto do tipo `Email`, o que nos poupou algumas linhas de código.

Criando um *mock* que lança uma exceção com muitos detalhes

No exemplo anterior, a exceção lançada era bem simples, bastando criar uma `AuthenticationFailedException` indicando sua mensagem de erro. E a mensagem que indicamos acabou se tornando literalmente a mensagem da exceção. Porém, algumas exceções não são tão simples assim. Vamos ver como testar outro possível problema em nosso serviço de envio de e-mails, a exceção `com.sun.mail.util.MailConnectException`.

Diferente da situação do tópico anterior, ela exige mais detalhes para ser criada e sua mensagem de erro não pode ser definida na criação. Vejamos como criar o *mock* para essa exceção no método de teste que também ficará na `EnvioEmailServiceTest`, cujo código está a seguir:

```
def 'deve lançar erro de conexão'() {
    given:
    def servidor = 'teste.com'
    def porta = 7777
    def timeout = 1

    this.email.send() >> {
        def sce = new SocketConnectException("", null, servidor, porta,
timeout)
        throw new EmailException(new MailConnectException(sce))
    }

    when:
    this.service.enviarEmails('assunto', 'conteudo', ['em@t.com'])

    then:
    def ex = thrown(EmailException)
    ex.cause.class == MailConnectException.class
    ex.cause.message.contains(servidor)
    ex.cause.message.contains(porta.toString())

    !this.service.enviados
}
```

No bloco `given` indicamos o servidor, porta e *timeout* que serão usados na criação do *mock* do teste. Em seguida, configuramos o método `send()` para lançar uma `EmailException` na qual definimos que a causa é uma `com.sun.mail.util.SocketConnectException`.

Para definir essa causa, não poderíamos usar um construtor que apenas define a mensagem, pois ele não existe na `SocketConnectException`. Tínhamos que definir uma **mensagem de detalhe** (que não se torna a mensagem da exceção, por isso ignoramos), uma **causa** (que ignoramos, por isso `null`), um **servidor**, uma **porta** e um **timeout**, estes três últimos definidos há pouco e que poderiam ter quaisquer valores válidos.

No bloco `when` apenas invocamos o método que queremos testar com quaisquer argumentos válidos. E no bloco `then` testamos:

- Se foi lançada uma `EmailException`;
- Se a causa da exceção foi uma `SocketConnectException`;
- Se a mensagem da causa contém o `servidor` e a `porta` definidos no bloco `given`. A mensagem da `SocketConnectException` é bem longa, mas contém o servidor e a porta que foram usados na tentativa de conexão com o servidor;
- Se o atributo `enviados` continua zerado.

7.5 Configurando retornos de métodos aninhados em mocks

Já vimos como configurar os retornos dos métodos de um *mock*. Porém, há situações em que precisamos de retornos de métodos de objetos que já foram obtidos de retornos de *mocks*. Um exemplo disso seria se precisássemos salvar os IMCs das pessoas em uma tabela de um banco de dados relacional e usássemos JPA. Imagine que a tabela para registro do IMC atual das pessoas tivesse sido mapeada pela classe `ImcAtual`, cujo código está a seguir:

```
package br.com.livrospockframework.capitulo07.entity;
import javax.persistence.Column;
import javax.persistence.Entity;

@Entity
public class ImcAtual {

    @Column(length=50)
    private String nome;

    @Column
    private double imc;
```

```
    public ImcAtual(String nome, double imc) {
        super();
        this.nome = nome;
        this.imc = imc;
    }
    // construtor padrão, getters e setters
```

E para salvar um único IMC por pessoa, criamos uma outra classe chamada `ImcAtualService`. Ela deve garantir que uma pessoa só possua um registro de IMC, sempre com o mesmo valor calculado. Seu código está a seguir:

```
package br.com.livrospockframework.capitulo07.services;

import java.sql.SQLException;
import javax.persistence.EntityManager;
import javax.persistence.NoResultException;
import br.com.livrospockframework.capitulo07.entity.ImcAtual;
import br.com.livrospockframework.capitulo07.model.ResultadoImc;

public class ImcAtualService {

    private static final String CONSULTA_IMC =
            "SELECT i FROM ImcAtual i WHERE nome = ?0";

    private EntityManager entityManager;

    private ImcAtual imcExistente(ResultadoImc resultado) {
        try {
            return this.entityManager
                    .createQuery(CONSULTA_IMC, ImcAtual.class)
                    .setParameter(0, resultado.getNome())
                    .getSingleResult();
        } catch (NoResultException e) {
            return null;
        }
    }
```

```
    public int registrarImc(List<ResultadoImc> resultados) throws
SQLException {

        int inseridos = 0;
        for (ResultadoImc resultado : resultados) {
            ImcAtual imcExistente = this.imcExistente(resultado);
            if (imcExistente != null) {
                imcExistente.setImc(resultado.getImc());
                continue;
            }
            ImcAtual imcAtual = new ImcAtual(resultado.getNome(),
resultado.getImc());
            this.entityManager.persist(imcAtual);
            inseridos++;
        }

        return inseridos;
    }

}
```

O método `imcExistente()` tenta retornar uma instância de `ImcAtual` a partir de um nome. Caso nenhum registro seja encontrado, o método retorna `null`. O método `registrarImc()` retorna o número de registros novos inseridos, porém atualiza os existentes. Ele invoca o `imcExistente()` e, caso encontre registro com o nome, apenas atualiza seu IMC. Caso contrário, cria um novo registro.

Essa classe seria fácil de testar não fosse um detalhe: seus métodos usam um atributo de instância privado do tipo `javax.persistence.EntityManager`, chamado `entityManager`. As dificuldades surgem devido aos seguintes fatos:

- Configurar uma `Persistence Unit` do JPA não é simples, pois exigiria uma série de dependências e configurações, além de tornar nosso teste dependente da disponibilidade e desempenho de um banco de dados. Como o objeto do tipo `EntityManager`

não é o foco do teste, podemos simplesmente criar um *mock* dele;
- No método `imcExistente()` usamos 3 métodos de forma aninhada sobre o `entityManager`: primeiro, `createQuery()`; sobre o retorno deste, `setParameter()`, e, sobre o retorno deste, `getSingleResult()`. Para conseguirmos testar essa situação, teremos que criar vários *mocks* e configurar seus retornos para que imitem essa chamada aninhada de métodos.

Vejamos agora como poderia ficar a classe de testes. Seguindo nosso padrão, vamos chamá-la de `ImcAtualServiceTest`. Vejamos seu código na sequência. Como ele ficaria um tanto grande, vamos transcrevê-lo e explicá-lo por partes.

```
package livrospock.capitulo07

import javax.persistence.TypedQuery
import br.com.livrospockframework.capitulo07.entity.ImcAtual
import br.com.livrospockframework.capitulo07.model.ResultadoImc
import import
br.com.livrospockframework.capitulo07.services.ImcAtualService
// demais imports

class ImcAtualServiceTest extends Specification {
    String nomeEncontrado = 'Zé Cadastrado'
    String nomeNaoEncontrado1 = 'João Não Cadastrado'
    String nomeNaoEncontrado2 = 'Maria Não Cadastrada'

    EntityManager entityManager
    ImcAtualService service

    def setup() {
        this.entityManager = Mock(EntityManager)

        def resultadoEncontrado = Mock(TypedQuery)
        def resultadoNaoEncontrado = Mock(TypedQuery)
        def instrucaoConsulta = Mock(TypedQuery)

        resultadoEncontrado.getSingleResult() >> new ImcAtual(nome:
```

```
this.nomeEncontrado)

        instrucaoConsulta.setParameter(0, this.nomeEncontrado) >>
resultadoEncontrado
        instrucaoConsulta.setParameter(0, this.nomeNaoEncontrado1) >>
resultadoNaoEncontrado
        instrucaoConsulta.setParameter(0, this.nomeNaoEncontrado2) >>
resultadoNaoEncontrado

        this.entityManager.createQuery(ImcAtualService.CONSULTA_IMC,
ImcAtual.class) >> instrucaoConsulta

        this.service = new
ImcAtualService(entityManager:this.entityManager)
    }
    // métodos de teste
}
```

Nesse código, temos apenas os atributos de instância, que serão usados em todos os testes, e o método `setup()`, que, como vimos anteriormente, será executado sempre antes de qualquer outro método de teste. Os atributos são:

- 3 nomes, sendo 1 para simular situações de nome já existente no banco de dados (`nomeEncontrado`) e 2 para simular nomes que não estão cadastrados (`nomeNaoEncontrado1` e `nomeNaoEncontrado2`);
- 1 objeto do tipo `EntityManager`, que será instanciado por um *mock*;
- 1 objeto do tipo `ImcAtualService`, que será um objeto real (não *mock*) usado nos testes dessa classe.

No método `setup()`, configuramos um *mock* simples para o `entityManager` e três *mocks* para objetos do tipo `javax.persistence.TypedQuery` (`resultadoEncontrado`, `resultadoNaoEncontrado` e `instrucaoConsulta`).

Na sequência, configuramos os *mocks* que serão usados de forma aninhada durante os testes:

- Método `getSingleResult()` a partir de `resultadoEncontrado`: retornará uma instância de `ImcAtual` com seu `nome` igual ao valor de `nomeEncontrado`;
- Método `instrucaoConsulta()`, com `0` e `nomeEncontrado` como argumentos, a partir de `instrucaoConsulta`: retornará o `resultadoEncontrado`;
- Método `instrucaoConsulta()`, com `0` e `nomeNaoEncontrado1` como argumentos, a partir de `instrucaoConsulta`: retornará o `resultadoNaoEncontrado`;
- Método `instrucaoConsulta()`, com `0` e `nomeNaoEncontrado2` como argumentos, a partir de `instrucaoConsulta`: retornará o `resultadoNaoEncontrado`;
- Método `createQuery()` a partir de `entityManager`: retornará o `instrucaoConsulta`.

Não foi necessário indicar o retorno do `getSingleResult()` a partir de `resultadoNaoEncontrado`. Como já explicado neste capítulo, métodos de *mocks* que não retornam primitivos retornam `null` por padrão.

A instrução final no método `setup()` é instanciar o `service` e preencher o seu `entityManager` com o *mock* do `EntityManager` recém-criado. Caso não saiba como funciona o construtor usado para instanciar um `ImcAtual` e um `ImcAtualService`, veja o item ***É possível atribuir valores de atributos na criação de objetos*** no *Apêndice A - Guia de Groovy para desenvolvedores Java*.

Com os *mocks* devidamente configurados, podemos testar o método privado `imcExistente()`, como no método de teste a seguir:

```
def 'deve retornar null ou o ImcAtual por nome'() {
    expect:
    !this.service.imcExistente(this.nomeNaoEncontrado1)
    !this.service.imcExistente(this.nomeNaoEncontrado2)
    this.service.imcExistente(this.nomeEncontrado)?.getNome() ==
this.nomeEncontrado
}
```

Como não havia necessidade de nenhum preparo adicional, esse método poderia conter apenas o bloco `expect`. Nesse bloco, verificamos se a invocação do método `imcExistente()`:

- Com os nomes que não deveriam ser encontrados pelo `EntityManager` retorna `null`;
- Com o nome que deveria ser encontrado pelo `EntityManager` retorna uma instância de `ImcAtual` cujo `nome` é igual ao atributo `nomeEncontrado`.

No próximo teste verificaremos se o método `registrarImc()` realmente só retorna a quantidade de registros inseridos. Vejamos esse teste no próximo código-fonte:

```
def 'deve inserir somente se ainda não existe'() {
    when:
    def resultadosE2 = [
        new ResultadoImc(nome:nomeEncontrado, imc: 25),
        new ResultadoImc(nome:nomeNaoEncontrado1, imc: 29),
        new ResultadoImc(nome:nomeNaoEncontrado2, imc: 35)
    ]

    def resultadosE1 = [
        new ResultadoImc(nome:nomeEncontrado, imc: 25),
        new ResultadoImc(nome:nomeNaoEncontrado1, imc: 29)
    ]

    def resultadosE0 = [
        new ResultadoImc(nome:nomeEncontrado, imc: 25)
    ]

    then:
    this.service.registrarImc(resultadosE2) == 2
    this.service.registrarImc(resultadosE1) == 1
    this.service.registrarImc(resultadosE0) == 0
}
```

Nesse teste, preparamos 3 listas de `ResultadoImc` que serão usadas nos testes no bloco `when`:

- `resultadosE2` : lista que contém 1 nome que deve ser encontrado e 2 que não devem;
- `resultadosE1` : lista que contém 1 nome que deve ser encontrado e outro que não;
- `resultadosE0` : lista que contém apenas 1 nome que deve ser encontrado.

No bloco `then` verificamos se a invocação do método `registrarImc()` retorna:

- Valor `2` quando invocado com `resultadosE2` ;
- Valor `1` quando invocado com `resultadosE1` ;
- Valor `0` quando invocado com `resultadosE0` .

7.6 Verificando a quantidade de execuções de métodos dos mocks

Uma das motivações para o uso de *mocks* expostas neste capítulo foi a de que há situações em que não precisamos que um componente faça sua ação real, e sim que seja solicitado a fazê-la. Mantendo a classe `ImcAtualService` como exemplo, imagine que queremos ter certeza de que o método `persist()` do `EntityManager` dela seja invocado uma certa quantidade de vezes durante a execução do método `registrarImc()` . Como os *mocks* do Spock também são ***spies***, é possível saber quantas vezes um método de um *mock* foi executado.

Não precisamos mexer em nada na `ImcAtualService` . Basta criarmos um novo teste na classe de testes `ImcAtualServiceTest` , que está no código-fonte a seguir:

```
def 'deve criar somente como novo por nome'() {
    given:
    def resultados = [
        new ResultadoImc(nome:nomeEncontrado, imc: 25),
        new ResultadoImc(nome:nomeNaoEncontrado1, imc: 29),
        new ResultadoImc(nome:nomeNaoEncontrado2, imc: 35)
    ]

    when:
    this.service.registrarImc(resultados)

    then:
    2 * this.entityManager.persist(_)
}
```

No cenário programado no último método de teste, criamos uma lista de 3 objetos do tipo `ResultadoImc`, em que 1 possui um nome já cadastrado e 2 não cadastrados. Portanto, o método `persist()` do `EntityManager` deveria ser invocado 2 vezes quando usamos o método `registrarImc()` usando essa lista. Nos blocos `given` e `when` não há novidades: configuração e execução do método de teste, respectivamente.

A grande novidade está no bloco `then`. A linha que existe nele passará no teste se o método `persist()` for invocado **2 vezes** com qualquer argumento durante a execução do código do bloco `when`. Isso foi possível porque o *mock* `entityManager` da classe de testes sempre é atribuído ao atributo de mesmo nome no objeto `service` da classe de testes no `setup()`.

É interessante destacar a peculiaridade e assertividade nos logs do Spock caso esse tipo de verificação falhe. Vamos supor que nosso teste esperasse (erroneamente) por 1 execução do método `persist()`. Nesse caso, o log de erro ficaria como o do log de exemplo a seguir:

```
FAILURE! - in livrospock.capitulo07.ImcAtualServiceTest
deveria criar somente como novo por
```

```
nome(livrospock.capitulo07.ImcAtualServiceTest)  Time elapsed: 0.01 sec
<<< FAILURE!
org.spockframework.mock.TooManyInvocationsError: Too many invocations for:

1 * this.entityManager.persist(_)   (2 invocations)
Matching invocations (ordered by last occurrence):
1 *
<EntityManager>.persist(br.com.livrospockframework.capitulo07.entity.ImcAt
ual@2bfc2f8b)  <-- this triggered the error
1 *
<EntityManager>.persist(br.com.livrospockframework.capitulo07.entity.ImcAt
ual@61853c7e)
```

Note que o Spock indica que houve mais invocações do que o esperado (*"Too many invocations"*), quantas invocações ocorreram e ainda os objetos envolvidos em cada execução. Caso a classe `ImcAtual` tivesse sobrescrito o `toString()` padrão do Java, o log poderia ter ficado ainda mais esclarecedor, podendo indicar, por exemplo, atributos dos objetos usados como argumento.

Conclusão

Neste capítulo, vimos como criar e configurar *mocks* com Spock para imitar o comportamento de componentes complexos e/ou perigosos que não são objetos do teste. No próximo capítulo, veremos como testar **REST APIs** com o Spock framework.

Capítulo 8
Testando RESTful APIs

Até aqui já vimos como criar vários tipos de testes e como trabalhar com *dublês* no Spock. Neste capítulo, veremos como criar testes para RESTful APIs, que é a tecnologia mais usada atualmente na integração de sistemas.

8.1 O que são RESTful APIs e a importância de serem bem testadas

Esta é uma breve introdução para o leitor ou a leitora que possui pouco ou nenhum conhecimento sobre essa tecnologia.

O REST (***RE****presentational* ***S****tate* ***T****ransfer*) é um estilo arquitetural que usa HTTP ou HTTPS como protocolo de comunicação. Foi proposto em 2000, na tese de doutorado de Roy Fielding, que, aliás, também é um dos pais do HTTP.

Quando uma API é criada adotando-se REST, dizemos que se trata de uma RESTful API. Porém, é muito comum vermos o termo **REST API** para o mesmo conceito.

Até meados de 2011, a arquitetura mais usada no mundo para implementar APIs era **SOAP**, mas, desde então, REST passou a ser mais adotada. Ross Mason, fundador da *MuleSoft*, escreveu um artigo falando sobre o fato de REST ter ultrapassado SOAP na integração de sistemas (veja em https://www.infoq.com/articles/rest-soap). Ao recorrermos ao *Google Trends*, veremos que o interesse ao longo do tempo sobre REST (termo de computação) ultrapassou o de SOAP (termo de

computação) exatamente em junho de 2011 e continuou crescendo, conforme mostra a figura a seguir:

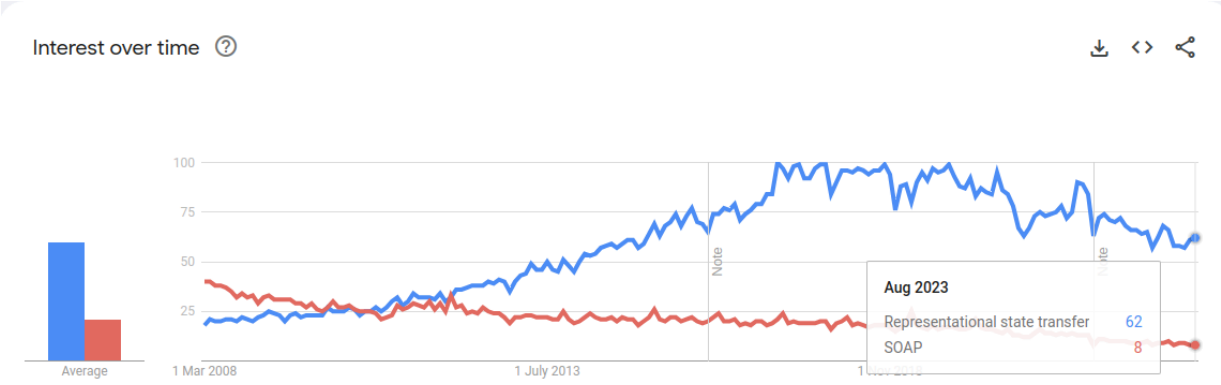


Figura 8.1: Pesquisas sobre REST x SOAP segundo o Google Trends. Fonte: Google Trends

PARA SABER MAIS SOBRE REST

Caso você queira se aprofundar nos conceitos e tecnologias de implementação de REST, há três livros da Casa do Código que abordam essa tecnologia:

- *Roadmap back-end - Conhecendo o protocolo HTTP e arquiteturas REST*, de Victor Osório;
- *Web Services REST com ASP .NET Web API e Windows Azure*, de Paulo Siécola;
- *Construindo APIs REST com Node.js*, de Caio Ribeiro Pereira.

Todas as principais plataformas de desenvolvimento da atualidade (Java, .Net, Python, JavaScript/Node.js, Go etc.) possuem frameworks e/ou bibliotecas para a implementação de RESTful APIs. E se olharmos as grandes empresas de tecnologia, como Google, Amazon, Microsoft, IBM, Facebook, operadoras de cartão de crédito etc., todas oferecem uma coleção de RESTful APIs para o consumo de seus serviços.

Isso tudo demonstra que, conforme Ross Mason antecipou em 2011, essa é a abordagem mais usada atualmente na integração de

sistemas. Assim, é muito provável que você tenha que escrever testes para alguma RESTful API interna e/ou de algum fornecedor. E caso você ou sua empresa tenha criado uma API, é imperativo que ela esteja bem testada para evitar que os que a consomem não tenham surpresas desagradáveis, como respostas em formato inesperado ou *endpoints* com endereço alterado de uma hora para outra, por exemplo.

Uma RESTful API pode ser implementada em praticamente qualquer linguagem de programação moderna. Porém, independentemente da plataforma de desenvolvimento usada na criação, trata-se de um serviço que usa o HTTP/HTTPS para enviar **requisições** (ou **solicitações**) e receber **respostas**.

Requisições HTTP

As requisições HTTP são usadas para que um cliente (ou *consumidor*) inicie uma comunicação com uma RESTful API. Uma requisição HTTP é composta basicamente de:

- **Método**: é um verbo que indica a natureza da chamada. Os métodos (ou métodos) HTTP mais usados em RESTful APIs são:
  - `GET` - Usado para recuperar um ou mais recursos;
  - `POST` - Usado para a criação de um recurso;
  - `PUT` - Usado para a atualização de um recurso;
  - `PATCH` - Parecido com o `PUT`, mas indicado quando a atualização de um recurso é parcial;
  - `DELETE` - Usado para excluir um recurso.

> Existem ainda vários outros métodos HTTP. Veja maiores detalhes nas páginas sobre métodos HTTP da especificação oficial do HTTP 1.1: https://tools.ietf.org/html/rfc7231#section-4 e https://tools.ietf.org/html/rfc5789#section-2.

- **Parâmetros**: um ou mais dados agrupados na forma de *chave x valor*. São usados para detalhar a operação indicada no

método. Por exemplo, em uma chamada com `GET`, os parâmetros podem servir para indicar os campos e valores envolvidos no filtro de uma consulta.

- **Cabeçalhos**: são muito parecidos com parâmetros, também sendo enviados agrupados em *chave x valor*. A diferença é que aqui são enviados dados que não possuem relação direta com a requisição. É muito comum serem usados para enviar as credenciais de acesso à API e o tipo de dado que está sendo enviado no **corpo**. Os cabeçalhos mais comuns são listados e descritos no *Mozilla Developers Network* (https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers).

- **Corpo**: é o conteúdo da requisição. Esse conteúdo pode ser um texto simples, um JSON, um XML ou até mesmo um conteúdo binário (PDF, ZIP etc.). Quando é enviado um corpo, o ideal é que também seja enviado um **cabeçalho** chamado *Content-type*. Alguns dos valores mais comuns para esse cabeçalho são: *application/json* (conteúdo JSON), *text/xml* (conteúdo XML) e *application/octet-stream* (conteúdo binário).

> **IMPORTANTE**: nem todos os métodos HTTP permitem o envio de um corpo. Dos citados aqui, apenas `POST`, `PUT` e `PATCH` permitem.

Respostas HTTP

As respostas HTTP são usadas pela RESTful API para devolver o resultado de uma requisição HTTP para um cliente. Uma resposta HTTP é composta basicamente de *status*, *cabeçalho* e *corpo*.

Status é um código numérico que indica o resultado geral da resposta. Existem vários códigos, divididos em famílias, sendo que as mais usadas em RESTful APIs são:

- **2xx** - Códigos de status que indicam que a requisição foi processada com sucesso. Entre os comuns estão:

    - *200 - Ok*: indica que deu tudo certo;
    - *201 - Created*: indica que um recurso foi criado com sucesso;
    - *202 - Accepted*: indica que a requisição foi aceita, mas que seu processamento pode estar sendo feito paralelamente.
- **4xx** - Códigos de status que indicam que a requisição foi feita com algo errado, ou seja, que o cliente não fez a requisição como deveria. Entre os comuns estão:
    - *400 - Bad request*: indica que alguma coisa nos parâmetros e/ou corpo da requisição está errada;
    - *401 - Unauthorized*: indica que era necessária alguma credencial de acesso, não fornecida;
    - *403 - Forbidden*: indica que um recurso não é permitido mesmo com as credenciais de acesso fornecidas;
    - *404 - Not found*: indica que o recurso solicitado não existe;
    - *405 - Method not allowed*: indica que o método HTTP é inválido para o *endpoint* chamado;
    - *415 - Unsupported Media Type*: indica que o tipo de corpo de requisição é inválido para o *endpoint* chamado.
- **5xx** - Códigos de status que indicam que a requisição não pode ser processada por problema no serviço. Entre os comuns estão:
    - *500 - Internal server error*: indica que ocorreu algum erro não tratado no serviço;
    - *503 - Service unavailable*: indica que o serviço está praticamente fora, só conseguindo responder para informar que está indisponível. É como receber esse status quando a API está sendo reiniciada.

Existem muitos outros status de resposta HTTP. Uma lista completa pode ser obtida no *Mozilla Developers Network* (https://developer.mozilla.org/en-US/docs/Web/HTTP/Status).

Cabeçalhos são valores agrupados em *chave x valor* que descrevem detalhes da resposta e do corpo, caso este esteja presente. O **corpo** é o conteúdo da resposta. Esse conteúdo pode ser um texto simples, um JSON, um XML ou até mesmo um conteúdo binário (PDF, ZIP etc.). Quando há um corpo na resposta, o ideal é que também seja enviado um **cabeçalho** chamado ***Content-type***. Alguns dos valores mais comuns para esse cabeçalho são: *application/json* (conteúdo JSON), *text/xml* (conteúdo XML) e *image/jpeg* (imagem).

Endpoints (ou chamadas)

Cada operação disponível em uma API é um ***endpoint*** (ou chamada). É para eles que enviamos as requisições HTTP e é deles que esperamos as respostas HTTP. Como exemplo simples de *endpoint*, podemos citar que uma API para a realização das operações básicas da aritmética deveria ter pelo menos 4 *endpoints* (ou chamadas): adição, subtração, multiplicação e divisão.

Clientes de uma RESTful API

Como o único requisito para consumir uma RESTful API é uma conexão HTTP/HTTPS, clientes desse tipo de serviço podem ser uma simples página HTML, um programa desktop, um aplicativo de smartphone ou até mesmo um processo em um sistema operacional. O cliente não tem a menor ideia de qual plataforma foi usada para desenvolver a API e pode ter sido criado em qualquer linguagem capaz de implementar um *cliente HTTP*. Esse modelo de comunicação está representado na figura a seguir:

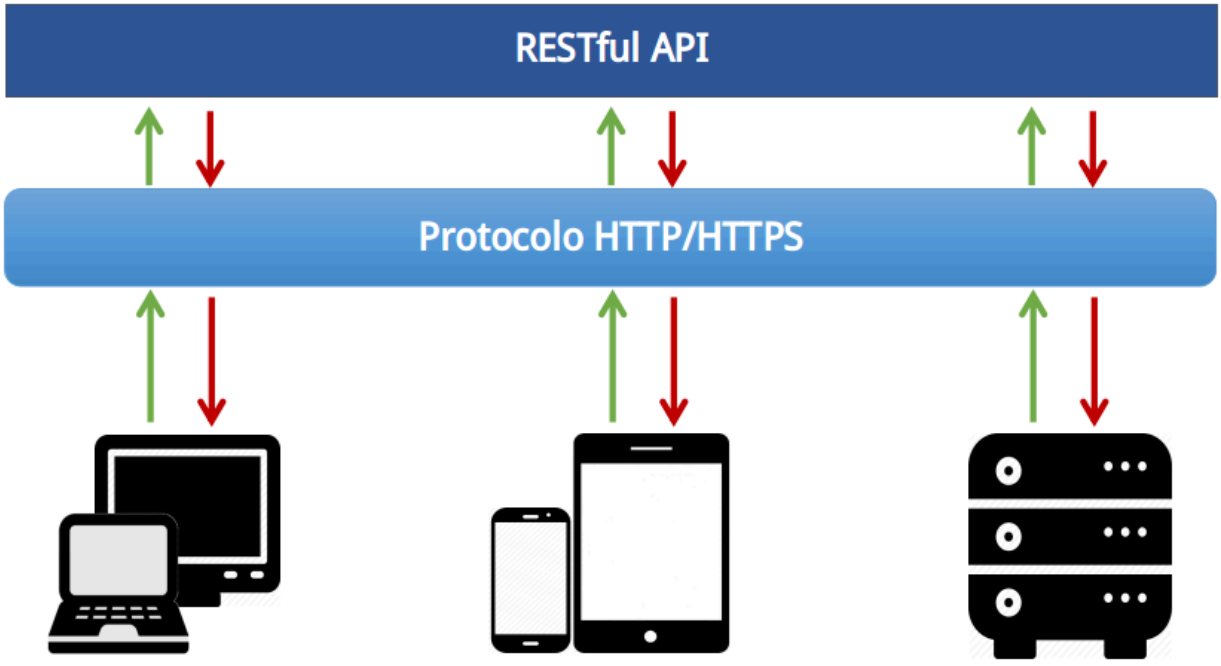


Figura 8.2: RESTful API comunicando-se com seus clientes por meio do HTTP

Como Groovy e Java permitem a criação de *clientes HTTP/HTTPS*, é possível criar testes para RESTful APIs com o Spock framework. É o que faremos a seguir.

8.2 Documentação da API de Cálculo e Banco de IMC

Vamos testar uma API de cálculo de IMC que armazena e permite excluir os cálculos solicitados. Ela possui 4 (quatro) *endpoints*, conforme mostra a figura a seguir:

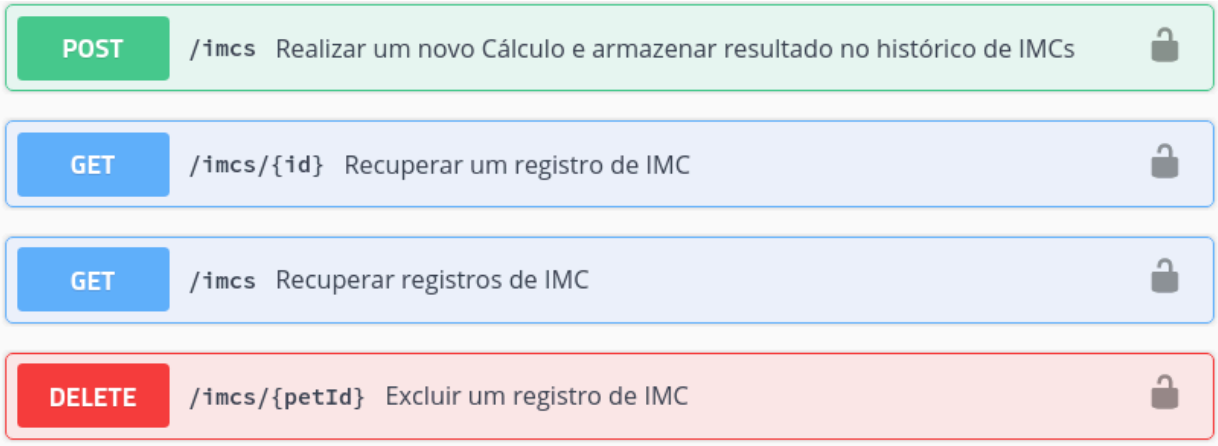


Figura 8.3: Endpoints da API de Cálculo e Banco de IMC

Vale lembrar que não é nosso foco criar uma RESTful API, vamos apenas descrever a API que será testada. Vejamos agora uma breve descrição das entradas e saídas de cada *endpoint*. Essas informações são necessárias para guiar nossos testes.

Endpoint `GET /imc/{id}`

Requisição

Nesse *endpoint*, o seguinte parâmetro deve ser enviado:

`id` **:** *Path param* com o identificador do cálculo solicitado.

Resposta com status 200 - OK

Quando o registro for encontrado com sucesso. Os seguintes cabeçalho e corpo de resposta são esperados:

Cabeçalho: `content-type: application/json`

Corpo da resposta:

```
{
  "id": ? (Inteiro),
  "imc": ? (Real),
  "nome": "?" (Alfanumérico),
  "peso": ? (Real),
```

```
  "altura": ? (Real),
  "condicao": "?" (Alfanumérico)
}
```

Resposta com status 404 - Not found

Quando a busca não encontra o registro solicitado.

Endpoint `GET /imcs`

Requisição

Nesse *endpoint*, o seguinte parâmetro deve ser enviado:

`apartir` **:** *Query param* com o valor mínimo de IMC para que um registro venha no resultado.

Resposta com status 200 - OK

Quando a solicitação encontrou pelo menos 1 registro.

Cabeçalho: `content-type: application/json`

Corpo da resposta:

```
[
  {
    "id": ? (Inteiro),
    "imc": ? (Real),
    "nome": "?" (Alfanumérico),
    "peso": ? (Real),
    "altura": ? (Real),
    "condicao": "?" (Alfanumérico)
  },
  {
    "id": ? (Inteiro),
    "imc": ? (Real),
    "nome": "?" (Alfanumérico),
    "peso": ? (Real),
    "altura": ? (Real),
    "condicao": "?" (Alfanumérico)
```

```
  },
  ...
]
```

Resposta com status 204 - No content

Quando a busca não traz nenhum resultado.

Endpoint `POST /imcs`

Requisição

Nesse *endpoint*, os seguintes cabeçalho e corpo de requisição devem ser enviados:

Cabeçalho: `content-type: application/json`

Corpo da requisição:

```
{
  "nome": "?" (Alfanumérico. Obrigatório. Mínimo de 2 letras),
  "peso": ? (Real. Obrigatório),
  "altura": ? (Real. Obrigatório)
}
```

Resposta com status 201 - Created

Quando a requisição é válida e o cálculo foi feito e registrado com sucesso. Os seguintes cabeçalho e corpo de resposta são esperados:

Cabeçalho: `content-type: application/json`

Corpo da resposta:

```
{
    "id": ? (Inteiro),
    "imc": ? (Real),
    "nome": "?" (Alfanumérico),
    "peso": ? (Real),
    "altura": ? (Real),
```

```
    "condicao": "?" (Alfanumérico)
}
```

Resposta com status 400 - Bad request

Quando nada for enviado no corpo da requisição ou o JSON enviado na requisição estiver inválido. Nesse caso, o corpo da resposta deve mencionar os campos inválidos.

Endpoint `DELETE /imcs/{id}`

Requisição

Nesse *endpoint*, o seguinte parâmetro deve ser enviado:

`id` **:** *Path param* com o identificador do cálculo solicitado.

Para que a chamada seja aceita, é necessário também enviar um cabeçalho de requisição:

`x-api-key` **:** *Cabeçalho* com a chave da API.

Resposta com status 204 - No content

Quando o registro solicitado foi excluído com sucesso. Nenhum corpo é esperado.

Resposta com status 404 - Not found

Quando o registro indicado para a exclusão não existe.

Resposta com status 401 - Unauthorized

Quando não é enviado o cabeçalho de chave de API.

Resposta com status 403 - Forbidden

Quando o cabeçalho de chave de API é inválido.

8.3 Testando o endpoint GET /imcs/{id}

Para testar o *endpoint* `GET /imcs/{id}`, vamos criar a classe de testes `ImcApiGetUmTest`. Os *imports* da classe ficam como no código a seguir:

```
import groovy.json.JsonSlurper
import livrospock.capitulo08.ImcResposta
import spock.lang.Specification

// imports estáticos para código mais "limpo"
import static java.net.http.HttpRequest.newBuilder
import static java.net.URI.create
import static java.net.http.HttpClient.newHttpClient
import static java.net.http.HttpResponse.BodyHandlers.ofString
```

Os *imports* estáticos foram criados para deixar o código com menos repetições, logo, mais simples. Será possível ver onde eles foram aplicados em vários dos trechos de código a seguir.

A classe `ImcResposta`, mencionada dos *imports*, foi criada para nos ajudar a validar as respostas da API. Segue seu código:

```
class ImcResposta {

    Long id
    String nome
    Double peso
    Double altura
    Double imc
    String condicao
}
```

O único atributo de instância dessa classe fica como no código a seguir:

```
def urlBase = 'http://localhost:8080/imcs'
```

O atributo `urlBase` contém a URL base usada em todos os cenários de teste da classe.

Testando uma requisição sem erros

Vamos criar um teste na classe `ImcApiGetUmTest`, que faz uma requisição sem erros e que verifica se a resposta está conforme o esperado para esse tipo de requisição. Vejamos esse teste a seguir:

```
def 'deve recuperar um registro de IMC'() {
    given: 'Id de IMC existente'
    def id = 1

    and: 'Criando a requisição'
    def requisicao = newBuilder()
                    .uri(create("${urlBase}/${id}"))
                    .GET()
                    .build()
    // newBuilder() e create() vêm de um dos imports estáticos

    when: 'Realizando a requisição e obtendo a resposta'
    def resposta = newHttpClient().send(requisicao, ofString())
    // newHttpClient() e ofString() vêm de um dos imports estáticos

    then: 'O status deve ser 200 e a resposta deve ser um JSON'
    resposta.statusCode() == 200
    resposta.headers().allValues("content-
type").contains("application/json")

    and: 'O corpo da resposta deve ser um JSON válido'
    Map mapa = new JsonSlurper().parseText(resposta.body()) // como a
resposta é um JSON, aqui será um Map
    assert new ImcResposta(mapa)
}
```

No bloco `given`, definimos o `id` de um IMC existente e, no bloco `and`, criamos um objeto que permite fazer uma chamada HTTP com método `GET`.

No bloco `when`, fazemos a requisição HTTP e obtemos sua resposta.

Usamos a primeira parte do bloco `then` para validar se o status que chegou foi `200 (Ok)`. Em seguida, verificamos se o `Content-type` da resposta foi `application/json`.

No bloco `and` (segunda parte do `then`), verificamos se o JSON do corpo da resposta possui a estrutura esperada. Primeiro convertemos o corpo da resposta em um `Map` que é utilizado para instanciar um objeto do tipo `ImcResposta`. Como essa classe possui a estrutura que esperamos que o JSON tenha, caso não ocorra erro na criação do objeto, o JSON estará válido.

Testando uma requisição informando um identificador inválido

Vamos criar agora um teste que realiza uma requisição com identificador propositalmente inexistente para o *endpoint* `GET /imcs/{id}` para verificar se a resposta está conforme a documentação. A seguir, o código do método desse teste:

```
def 'deve retornar 404 e sem corpo em caso de id inexistente'() {
    given: 'Id inexistente'
    def idInexistente = -9

    and: 'Criando a requisição'
    def requisicao = newBuilder()
            .uri(create("${urlBase}/${idInexistente}"))
            .GET()
            .build()
    // newBuilder() e create() vêm de um dos imports estáticos

    when: 'Realizando a requisição e obtendo a resposta'
    def resposta = newHttpClient().send(requisicao, ofString())
    // newHttpClient() e ofString() vêm de um dos imports estáticos

    then: 'O status deve ser 404 e o corpo da resposta deve estar vazio'
    resposta.statusCode() == 404
```

```
    resposta.body().isBlank() // ou, apenas: !resposta.body()
}
```

No bloco `given`, definimos o `id` de um IMC inexistente e, no bloco `and`, criamos um objeto que permite fazer uma chamada HTTP com método `GET`.

No bloco `when`, fazemos a requisição HTTP e obtemos sua resposta.

Usamos o bloco `then` para verificar se a resposta chegará com o status `404 (Not found)` e com o corpo vazio.

8.4 Testando o endpoint GET /imcs

Para testar o *endpoint* `GET /imcs/{id}`, vamos criar a classe de testes `ImcApiListaTest`, que é muito parecida com a `ImcApiGetUmTest`. Os *imports* e o atributo de instância `urlBase` são exatamente os mesmos.

Testando uma requisição sem erros que solicita todos os cálculos de IMC

Vamos criar um teste na classe `ImcApiGetListaTest` que faz uma requisição sem erros e que verifica se a resposta está conforme o esperado para esse tipo de requisição. Vejamos esse teste a seguir:

```
def 'deve recuperar lista de todos os IMCs'() {
    given: 'Criando a requisição'
    def requisicao = newBuilder()
            .uri(create(urlBase))
            .GET()
            .build()
    // newBuilder() e create() vêm de um dos imports estáticos

    when: 'Realizando a requisição e obtendo a resposta'
    def resposta = newHttpClient().send(requisicao, ofString())
```

```
    // newHttpClient() e ofString() vêm de um dos imports estáticos

    then: 'O status deve ser 200 e a resposta deve ser um JSON'
    resposta.statusCode() == 200
    resposta.headers().allValues("content-
type").contains("application/json")

    and: 'O corpo da resposta deve ser uma lista de JSONs válidos'
    List<Map> lista = new JsonSlurper().parseText(resposta.body()) // como
a resposta é um vetor de JSONs, aqui será uma List de Map
    lista.size() > 0
    lista.each {
        assert new ImcResposta(it)
    }
}
```

No bloco `given`, criamos um objeto que permite fazer uma chamada HTTP com método `GET`.

No bloco `when`, fazemos a requisição HTTP e obtemos sua resposta.

Usamos a primeira parte do bloco `then` para validar se o status que chegou foi `200 (Ok)`. Em seguida, verificamos se o `Content-type` da resposta foi `application/json`.

No bloco `and` (segunda parte do `then`), verificamos se o JSON do corpo da resposta possui a estrutura esperada. Primeiro convertemos o corpo da resposta em uma `List` de `Map` e usamos cada elemento nela para instanciar um objeto do tipo `ImcResposta`. Como essa classe possui a estrutura que esperamos que o JSON tenha, caso não ocorra erro na criação do objeto, o JSON estará válido.

Dentro de qualquer tipo de loop as asserções devem iniciar com `assert` para que a mensagem de erro no log de execução dos testes seja mais clara.

Testando uma requisição sem erros que não retorna nenhum cálculo de IMC

Vamos criar um teste na classe `ImcApiGetListaTest` que faz uma requisição com o parâmetro `apartir` sem erros, mas com um valor que não trará nenhum resultado, e que verifica se a resposta está conforme o esperado para esse tipo de requisição. Vejamos esse teste a seguir:

```
def 'deve retornar 204 e sem corpo caso o filtro não encontre
resultados'() {
    given: 'Parâmetro "apartir" impossível de existir'
    def imcImpossivel = 200

    and: 'Criando a requisição'
    def requisicao = newBuilder()
                    .uri(create("${urlBase}?apartir=${imcImpossivel}"))
                    .GET()
                    .build()

    when: 'Realizando a requisição e obtendo a resposta'
    def resposta = newHttpClient().send(requisicao, ofString())

    then: 'O status deve ser 204 e o corpo da resposta deve estar vazio'
    resposta.statusCode() == 204
    resposta.body().isBlank() // ou, apenas: !resposta.body()
}
```

No bloco `given`, definimos o `imcImpossivel` com um valor de IMC impossível de existir e, no bloco `and`, criamos um objeto que permite fazer uma chamada HTTP com método `GET`.

No bloco `when`, fazemos a requisição HTTP e obtemos sua resposta.

Usamos o bloco `then` para verificar se a resposta chegará com o status `204 (No content)` e com o corpo vazio.

8.5 Testando o endpoint POST /imcs

Vamos criar uma classe de testes chamada `ImcApiPostTest` e testar as situações de requisição perfeita e com problemas no JSON enviado.

Os *imports* são os mesmos dos testes anteriores, mais estes dois que estão no código a seguir:

```
import static groovy.json.JsonOutput.toJson
import static java.net.http.HttpRequest.BodyPublishers.noBody
```

Além do atributo `urlBase`, há um novo, como pode ser visto no próximo código:

```
def corpoRespostaString = HttpResponse.BodyHandlers.ofString()
```

O objeto `corpoRespostaString` será usado quando fizermos as chamadas aos *endpoints*. Sua criação foi necessária porque já existia uma importação estática de um método `ofString()` (`import static java.net.http.HttpRequest.BodyPublishers.ofString`).

Testando uma requisição sem erros

Vamos criar um teste na classe `ImcApiPostTest` que faz uma requisição sem erros e que verifica se a resposta está conforme o esperado para esse tipo de requisição. Vejamos esse teste a seguir:

```
def 'deve registar e calcular IMC'() {
    given: 'Requisição com JSON válido'
    def corpoRequisicao = [
        nome: 'Maria Gentil',
        peso: 70.00,
        altura: 1.71
    ]

    and: 'IMC e condição desejadas ao final da requisição'
    def imcDesejado = 23.94
    def condicaoDesejada = 'Peso Normal'

    and: 'Criando a requisição'
    def requisicao = newBuilder()
                    .uri(create(urlBase))
```

```
                    .POST(ofString(toJson(corpoRequisicao)))
                    .headers('content-type', 'application/json')
                    .build()
    // newBuilder(), create(), ofString() e toJson() vêm de um dos imports
estáticos

    when: 'Realizando a requisição e obtendo a resposta'
    def resposta = newHttpClient().send(requisicao, corpoRespostaString)
    // newHttpClient() vem de um dos imports estáticos

    then: 'O status deve ser 201 e a resposta deve ser um JSON'
    resposta.statusCode() == 201
    resposta.headers().allValues('content-
type').contains('application/json')

    and: 'O corpo da resposta deve ser um JSON válido'
    !resposta.body().isBlank() // ou, apenas: resposta.body()
    def corpoResposta = new JsonSlurper().parseText(resposta.body())
    corpoResposta.nome == corpoRequisicao.nome
    corpoResposta.peso == corpoRequisicao.peso
    corpoResposta.altura == corpoRequisicao.altura
    corpoResposta.imc.round(2) == imcDesejado
    corpoResposta.condicao == condicaoDesejada
}
```

No bloco `given`, criamos um `Map` com valores válidos para uma requisição para esse *endpoint* e, no bloco `and`, criamos um objeto que permite fazer uma chamada HTTP com método `POST`.

No bloco `when`, fazemos a requisição HTTP e obtemos sua resposta.

Usamos a primeira parte do bloco `then` para verificar se a resposta chegará com o status `201 (Created)` e com o cabeçalho `content-type` igual a `application/json`.

Na segunda parte (bloco `and`), verificamos se o corpo da resposta não está vazio e se sua estrutura está conforme o esperado, verificando campo por campo.

Testando uma requisição sem enviar o corpo de requisição

Vamos criar agora um teste que realiza uma requisição propositalmente malfeita para o *endpoint* `POST /imcs` — sem enviar nada no corpo de requisição — para verificar se a resposta está conforme especificado. A seguir, o código do método desse teste:

```
def 'deve receber 400 ao tentar criar IMC com JSON ausente'() {
    given: 'Requisição sem corpo'
    def requisicao = newBuilder()
                     .uri(create(urlBase))
                     .POST(noBody())
                     .build()
    // newBuilder(), create() e noBody() vêm de um dos imports estáticos

    when: 'Realizando a requisição e obtendo a resposta'
    def resposta = newHttpClient().send(requisicao, corpoRespostaString)
    // newHttpClient() vem de um dos imports estáticos

    then: 'O status deve ser 400'
    resposta.statusCode() == 400
}
```

No bloco `given`, criamos um objeto que permite fazer uma chamada HTTP com método `POST` indicando que não haverá corpo da requisição (método `noBody()` dentro de `POST()`).

No bloco `when`, fazemos a requisição HTTP e obtemos sua resposta.

Usamos a primeira parte do bloco `then` para verificar se a resposta chegará com o status `400 (Bad request)`.

Testando uma requisição enviando um corpo de requisição inválido

Vamos criar agora um teste que realiza uma requisição propositalmente malfeita para o *endpoint* `POST /imcs` — corpo de requisição com valores inválidos — para verificar se a resposta está conforme especificado. A seguir, o código do método desse teste:

```
def 'deve receber 400 ao tentar criar IMC com JSON com campos inválidos'()
{
    given: 'Corpo de requisição com JSON inválido (nome, peso e altura
inválidos)'
    def corpoRequisicao = [
        nome: 'Z',
        peso: -98.21,
        altura: 0.00
    ]

    def requisicao = newBuilder()
                    .uri(create(urlBase))
                    .POST(ofString(toJson(corpoRequisicao)))
                    .headers('content-type', 'application/json')
                    .build()
    // newBuilder(), create(), ofString() e toJson() vêm de um dos imports
estáticos

    when: 'Realizando a requisição e obtendo a resposta'
    def resposta = newHttpClient().send(requisicao, corpoRespostaString)
    // newHttpClient() vem de um dos imports estáticos

    then: 'O status deve ser 400'
    resposta.statusCode() == 400

    and: 'O corpo da resposta deve ser um JSON válido'
    def corpoResposta = resposta.body()
    corpoResposta.contains('Error count: 3') // total de erros
    corpoResposta.contains('imc.nome') // nome do imc é mencionado na
mensagem de erro
    corpoResposta.contains('imc.peso') // peso do imc é mencionado na
mensagem de erro
    corpoResposta.contains('imc.altura') // altura do imc é mencionada na
mensagem de erro
}
```

No bloco `given`, criamos um `Map` que será usado na requisição e, no bloco `and`, criamos um objeto que permite fazer uma chamada HTTP com método `POST` com corpo da requisição com valores inválidos.

No bloco `when`, fazemos a requisição HTTP e obtemos sua resposta.

Usamos a primeira parte do bloco `then` para verificar se a resposta chegará com o status `400 (Bad request)`. Na segunda parte (bloco `and`), verificamos se o corpo da resposta menciona que existem 3 erros e os nomes dos campos com erro.

8.6 Testando o endpoint DELETE /imcs/{id}

Para testar o *endpoint* `DELETE /imcs/{id}`, vamos criar a classe de testes `ImcApiDeleteTest`. Seus *imports* podem ser os mesmos da `ImcApiGetUmTest`. O atributo `urlBase` também deve existir, como nos testes anteriores. Vamos testar apenas 3 cenários:

1. Exclusão ocorrida com sucesso;
2. Erro por não enviar chave de API;
3. Erro por enviar chave de API inválida.

Testando uma requisição sem erros

Vamos criar um teste na classe `ImcApiDeleteTest` que faz uma requisição sem erros e que verifica se a resposta está conforme o esperado para esse tipo de requisição. Vejamos esse teste a seguir:

```
def 'deve excluir um registro de IMC caso o Id e a chave de API sejam válidos'() {
    given: 'Id de IMC existente'
    def id = 1

    and: 'Criando a requisição'
    def requisicao = newBuilder()
                    .uri(create("${urlBase}/${id}"))
                    .DELETE()
                    .headers('x-api-key','chave-valida')
                    .build()
    // newBuilder() e create() vêm de um dos imports estáticos
```

```
    when: 'Realizando a requisição e obtendo a resposta'
    def resposta = newHttpClient().send(requisicao, ofString())
    // newHttpClient() vem de um dos imports estáticos

    then: 'O status deve ser 204 e a resposta deve estar sem corpo'
    resposta.statusCode() == 204
    resposta.body().isBlank() // ou, apenas: !resposta.body()
}
```

No bloco `given`, definimos o `id` de um IMC existente e, no bloco `and`, criamos um objeto que permite fazer uma chamada HTTP com método `DELETE` e enviando uma chave de API válida no cabeçalho `x-api-key`.

No bloco `when`, fazemos a requisição HTTP e obtemos sua resposta.

Usamos o bloco `then` para validar se o status que chegou foi `204 (No content)` e se o corpo da resposta está vazio.

Testando uma requisição sem enviar a chave da API

Vamos criar agora um teste que realiza uma requisição sem enviar a chave da API para o *endpoint* `DELETE /imcs/{id}` para verificar se a resposta está conforme especificado. A seguir, o código do método desse teste:

```
def 'deve retornar 401 caso a chave de API não seja enviada'() {
    given: 'Id de IMC existente'
    def id = 1

    and: 'Criando a requisição'
    def requisicao = newBuilder()
                     .uri(create("${urlBase}/${id}"))
                     .DELETE()
                     .build()

    when: 'Realizando a requisição e obtendo a resposta'
    def resposta = newHttpClient().send(requisicao, ofString())
```

```
    then: 'O status deve ser 401'
    resposta.statusCode() == 401
}
```

No bloco `given`, definimos o `id` de um IMC existente e, no bloco `and`, criamos um objeto que permite fazer uma chamada HTTP com método `DELETE`, mas sem enviar a chave de API, pois não enviamos o cabeçalho `x-api-key`.

No bloco `when`, fazemos a requisição HTTP e obtemos sua resposta.

Usamos o bloco `then` apenas para verificar se a resposta chegará com o status `401 (Unauthorized)`.

Testando uma requisição enviando uma chave da API inválida

Vamos criar agora um teste que realiza uma requisição enviando uma chave da API inválida para o *endpoint* `DELETE /imcs/{id}` para verificar se a resposta está conforme especificado. A seguir, o código do método desse teste:

```
def 'deve retornar 403 caso a chave de API seja inválida'() {
    given: 'Id de IMC existente'
    def id = 1

    and: 'Criando a requisição'
    def requisicao = newBuilder()
                    .uri(create("${urlBase}/${id}"))
                    .DELETE()
                    .headers('x-api-key','chave-NÚTIL')
                    .build()

    when: 'Realizando a requisição e obtendo a resposta'
    def resposta = newHttpClient().send(requisicao, ofString())

    then: 'O status deve ser 403'
```

```
    resposta.statusCode() == 403
}
```

No bloco `given`, definimos o `id` de um IMC existente e, no bloco `and`, criamos um objeto que permite fazer uma chamada HTTP com método `DELETE` e enviando uma chave de API no cabeçalho `x-api-key`, porém com valor inválido.

No bloco `when`, fazemos a requisição HTTP e obtemos sua resposta.

Usamos o bloco `then` apenas para verificar se a resposta chegará com o status `403 (Forbidden)`.

Conclusão

Neste capítulo, vimos como criar testes funcionais para RESTful APIs com Spock. No próximo capítulo, veremos algumas **funcionalidades avançadas do framework** que ajudam a resolver vários pequenos problemas no cotidiano da execução de testes automatizados.

Capítulo 9
Recursos avançados

Com o conteúdo que já estudamos até aqui, já é possível criar qualquer tipo de teste dentre os quais o Spock se propõe a implementar. Porém, para situações pontuais que poderiam exigir *workarounds* e/ou repetição de código, este capítulo apresenta alguns recursos avançados do Spock.

9.1 Ignorando testes

Pode ser que você precise que alguns dos testes automatizados que criou sejam ignorados. Mas por que eu criaria um teste para ignorá-lo? Existem várias situações que exigem isso, dentre as quais se destacam:

- Uma funcionalidade que estava funcional e testada ficou desabilitada por um tempo, mas voltará a funcionar. Enquanto isso, ignoramos seu teste;
- Foi adotada a técnica **TDD** (falaremos um pouco dela logo mais), o que exige que o teste seja criado antes da funcionalidade. Portanto, seu teste deve ser ignorado se falhar;
- Uma funcionalidade só foi projetada para funcionar em determinadas condições de ambiente (sistema operacional, propriedades de sistema etc.). Logo, seus testes devem ser ignorados quando executados fora dessas condições.

Caso seja necessário simplesmente ignorar a execução de determinados testes, poderíamos simplesmente comentar ou mesmo excluir o código do teste. Mas essas são medidas pouco profissionais. O Spock possui anotações que permitem manter o

código do teste na classe, porém ignorando sua execução total ou parcialmente.

Ignorando testes com `@Ignore`

A anotação `@spock.lang.Ignore` é usada para que o Spock ignore a execução de um método ou de toda uma classe de testes. Essa anotação aceita, opcionalmente, um valor `String` para que seja descrito o motivo por que a classe ou teste está sendo ignorado.

Quando **usada sobre a definição da classe**, ela faz com que **todos os seus métodos de teste sejam ignorados**. A classe simplesmente é ignorada na fase de testes do Maven. Por exemplo, vamos supor que precisemos ignorar todos os testes da classe `livrospock.capitulo08.ImcApiPostTest` do capítulo anterior. Basta usar a `@Ignore` como no código a seguir:

```
package livrospock.capitulo08

import spock.lang.Ignore

@Ignore
class ImcApiPostTest extends Specification {
...
}
```

Se quisermos indicar o motivo de estarmos ignorando a classe, basta pôr um valor `String` na anotação `@Ignore`, como no próximo exemplo:

```
@Ignore('O Endpoint POST /imcs/calculo está desabilitado até 2019')
class ImcApiPostTest extends Specification {
```

Quando **usada sobre a assinatura de um método**, ela faz com que **o método anotado seja ignorado** quando a classe de teste for executada. O método não é executado na fase de testes do Maven. Por exemplo, vamos supor que precisemos ignorar um dos testes da classe `livrospock.capitulo08.ImcApiPostTest` do capítulo

anterior. Basta usar a `@Ignore` sobre o método desejado, como no código a seguir:

```
class ImcApiPostTest extends Specification {

    @Ignore
    def 'deve receber 401 ao tentar calcular IMC sem informar a chave da API'() {
        ...
    }
}
```

Com a configuração feita no último código, o método `'deve receber 401 ao tentar calcular IMC sem informar a chave da API'()` não será executado na fase de testes do Maven.

E assim como quando usamos essa anotação sobre a classe, também podemos indicar sobre um método o motivo de o teste estar sendo ignorado, como no próximo código:

```
class ImcApiPostTest extends Specification {

    @Ignore('Aparentemente a API não devolve mais 401, só 403')
    def 'deve receber 401 ao tentar calcular IMC sem informar a chave da API'() {
        ...
    }
}
```

Ignorando testes com `@IgnoreRest`

Imagine que uma classe de testes precise que todos os métodos, menos um, sejam ignorados. Isso pode ocorrer, por exemplo, quando um conjunto de funcionalidades é refatorado, deixando-as suspensas por um tempo. Nesse caso, em vez de usar a `@Ignore` sobre todos os métodos menos um, podemos simplesmente anotar apenas um método com `@spock.lang.IgnoreRest`. Essa anotação não possui nenhum atributo.

No código de exemplo a seguir, solicitamos ao Spock que ignore todos os testes da classe `livrospock.capitulo08.ImcApiGetUmTest` do capítulo anterior, menos o `'deve recuperar um IMC'()`:

```
package livrospock.capitulo08

import @spock.lang.IgnoreRest

class ImcApiGetUmTest extends Specification {

    @IgnoreRest
    def 'deve recuperar um IMC'() {
     ...
    }

    // demais métodos de teste
}
```

Com a configuração feita no último código, quando o Maven testar a classe `ImcApiGetUmTest`, apenas o método `'deve recuperar um IMC'()` será executado, não importa quantos métodos mais de teste existam na classe.

É possível usar a `@IgnoreRest` em mais de um método?

Sim, é possível. Supondo que uma classe de testes tenha 10 (dez) métodos e usemos a `@IgnoreRest` sobre 2 (dois) deles, estes serão os únicos a serem executados.

Ignorando resultado de testes com `@PendingFeature`

A anotação `@spock.lang.PendingFeature` só pode ser usada em métodos e indica que o teste **será executado**, porém que **deve falhar**. Ou seja, caso um teste anotado com ela passe, o Spock vai esperar que o teste falhe para marcá-lo como `passed`.

Atenção! Essa anotação não ignora o teste. Ele sempre será executado se anotado com ela!

Como o nome sugere, essa anotação deve estar presente em testes de funcionalidades que ainda não estão prontas, por isso esse comportamento de esperar que o teste falhe. Quando se trabalha com a metodologia **TDD**, é comum a criação de testes antes da própria funcionalidade.

> **O QUE É TDD?**
>
> É uma técnica proposta por Kent Beck em 2003 em seu clássico livro *Test-Driven Development: By Example*. Nela, o ciclo de desenvolvimento deve ser:
>
> **1.** Crie um teste antes da funcionalidade.
>
> **2.** Execute o teste. Ele vai, lógico, falhar no início (lembre-se de que a funcionalidade ainda não foi criada). Aí então é que você...
>
> **3.** Escreve o código da funcionalidade.
>
> **4.** Execute os testes automatizados e veja se executam com sucesso.
>
> **5.** Refatore o código.
>
> Kent defende que essa técnica reduz consideravelmente a execução de *debugs* e força a pessoa desenvolvedora a conhecer precocemente e profundamente o código que vai criar ou evoluir.

Para exemplificar, vamos supor que um dos testes da classe `livrospock.capitulo08.ImcApiDeleteTest` do capítulo anterior precise necessariamente falhar por ainda se tratar de um teste de uma funcionalidade que não deveria estar funcionando.

```
package livrospock.capitulo08

import @spock.lang.PendingFeature

class ImcApiDeleteTest extends Specification {
```

```
    @PendingFeature
    def 'deve excluir um registro de IMC'() {
       // verificação que faz o teste NÃO passar
    }

    // demais métodos de teste
}
```

No último código, o teste é executado e, caso o teste falhe, o Spock fará com que o Maven veja o teste como `passed`. Porém, caso o teste passe, aí sim o Maven vai se comportar como teste que falhou. Nesse caso, o log de erro seria como o do exemplo a seguir:

```
Failed tests:
  deveria excluir um registro de IMC Feature is marked with
@PendingFeature but passes unexpectedly
```

A ideia dessa anotação é deixar o teste pronto para o caso de funcionalidade não concluída. Assim, quando a funcionalidade ficar pronta, basta tirar a anotação `@PendingFeature`.

Não era melhor simplesmente pôr uma `@Ignore` sobre o teste?

A `@Ignore` simplesmente ignora a execução do teste. A `@PendingFeature` é para quando queremos realmente ter certeza de que a funcionalidade ainda não existe. E só é possível ter essa certeza testando-a.

Ignorando testes com `@IgnoreIf`

Vimos como ignorar testes, mas, e se para ignorar um teste precisarmos de uma condição? Um exemplo disso seria verificar se o código testado está sendo testado em um ambiente **homologado**, ou seja, se determinadas configurações do ambiente estão adequadas ao sistema. É possível que determinada funcionalidade de um sistema não seja compatível com o sistema operacional da família Windows ou com o Java 7 ou com determinado *kernel* do

Linux. Para situações como essa, existe a anotação `@spock.lang.IgnoreIf`.

A anotação `@IgnoreIf` pode ser usada sobre a classe ou sobre um método de teste. Ela sempre espera uma *closure* cujo conteúdo será usado para avaliar se o teste deve ou não ser ignorado. Seu uso seria como no código de exemplo a seguir:

```
import spock.lang.IgnoreIf

class AlgumaClasseTest extends Specification {

    @IgnoreIf({os.windows})
    def 'cenário não executado em Windows'() {
     ...
    }

}
```

Na anotação `@IgnoreIf` desse código, configuramos que o método de teste anotado será ignorado caso o sistema operacional (SO) seja da família Windows. Mas, o que é esse `os`?

No Spock, existem algumas classes utilitárias que facilitam a recuperação de várias informações do ambiente onde os testes estão sendo executados. São elas:

- `spock.util.environment.OperatingSystem`: classe que possui métodos que retornam informações sobre o **sistema operacional (SO)** onde o teste está sendo executado;
- `spock.util.environment.Jvm`: classe que possui métodos que retornam informações sobre a **JVM** onde o teste está sendo executado.

Para facilitar o uso dessas classes utilitárias `OperatingSystem` e `Jvm`, existem atalhos, descritos a seguir:

`os` - Atalho para `spock.util.environment.OperatingSystem`

- `os.name` : nome completo;
- `os.version` : versão;
- `os.family` : família (*Windows*, *Linux*, *macOs*, *Solaris* ou *Other*);
- `os.windows` : verdadeiro se o SO for da família *Windows*;
- `os.linux` : verdadeiro se o SO for da família *Linux*;
- `os.macOs` : verdadeiro se o SO for da família *macOs*;
- `os.solaris` : verdadeiro se o SO for da família *Solaris*;
- `os.other` : verdadeiro se o SO não for de nenhuma das famílias anteriores.

Veja alguns exemplos de uso:

- Teste que deve ser ignorado se o SO for um *Linux*:

```
@IgnoreIf({os.linux})
def 'cenário não executado em Linux'() {
...
}
```

- Teste que deve ser ignorado se o SO for um *Linux* ou *Solaris*:

```
@IgnoreIf({os.linux || os.solaris})
def 'cenário não executado em Linux nem Solaris'() {
...
}
```

- Teste que deve ser ignorado se a versão do SO for a *4.13.0-32-generic*:

```
@IgnoreIf({os.version == '4.13.0-32-generic' })
def 'cenário não executado em SO de versão 4.13.0-32-generic'() {
...
}
```

`jvm` - Atalho para `spock.util.environment.Jvm`

- `jvm.javaSpecificationVersion` : retorna a versão de especificação da JVM (*1.6*, *1.7*, *1.8* etc.);
- `jvm.javaVersion` : retorna a versão da JVM (*1.8.0_151*);
- `jvm.java5` : verdadeiro se a JVM for da família *Java 5*;

- `jvm.java5Compatible` : verdadeiro se a JVM for compatível com *Java 5*;
- `jvm.java6` : verdadeiro se a JVM for da família *Java 6*;
- `jvm.java6Compatible` : verdadeiro se a JVM for compatível com *Java 6*;
- `jvm.java7` : verdadeiro se a JVM for da família *Java 7*;
- `jvm.java7Compatible` : verdadeiro se a JVM for compatível com *Java 7*;
- `jvm.java8` : verdadeiro se a JVM for da família *Java 8*;
- `jvm.java8Compatible` : verdadeiro se a JVM for compatível com *Java 8*;
- `jvm.java9` : verdadeiro se a JVM for da família *Java 9*;
- `jvm.java9Compatible` : verdadeiro se a JVM for compatível com *Java 9*.
    - Exemplo de uso: teste que deve ser ignorado se a JVM da execução não for compatível com *Java 7*:

```
@IgnoreIf({!jvm.java7Compatible})
def 'cenário ignorado em JVM não compatível com Java 7'() {
...
}
```

Caso você queira sujeitar a execução ou não de um teste a propriedades do sistema ou informações de ambiente, é possível usar os atalhos a seguir:

- **`sys` : atalho para `System.getProperty(?)`**
    - Exemplo de uso: teste que deve ser ignorado caso o país do usuário seja os *Estados Unidos* (`us`):

```
@IgnoreIf({sys['user.country']=='US'})
def 'cenário não executado caso o país do usuário seja US'() {
...
}
```

- **`env` - atalho para `System.getenv(?)`**

- Exemplo de uso: teste que deve ser ignorado caso o *SHELL* do SO seja `/bin/bash`:

```
@IgnoreIf({env['SHELL']=='/bin/bash'})
def 'cenário não executado se o SHELL do SO for /bin/bash'() {
...
}
```

Indicando quando um teste deve ser executado com `@Requires`

A anotação `@spock.lang.Requires` funciona de forma muito parecida com a `@spock.lang.IgnoreIf`. A diferença é que a condição que se coloca na anotação indica o motivo de o teste ser **executado**, e não ignorado.

Por exemplo, vamos supor que precisemos que um teste deva ser executado somente se o SO for um *Linux*:

```
@Requires({os.linux})
def 'cenário que executa somente em Linux'() {
...
}
```

9.2 Tolerância de tempo de um teste com @Timeout

Algumas funcionalidades têm como requisito funcional um limite de tempo para execução. Um exemplo disso seria o tempo que um caixa eletrônico deve esperar até que consiga expelir o dinheiro solicitado em um saque. Com o Spock, por meio do uso da anotação `@spock.lang.Timeout`, é possível fazer com que um teste falhe automaticamente caso leve mais que um determinado tempo para concluir sua execução.

Essa anotação pode estar sobre a classe, o que indica um tempo de tolerância de execução para todos os seus testes. Ela também pode estar sobre a assinatura de apenas um método, o que indica a tolerância de tempo apenas de seu teste.

No código de exemplo a seguir, vamos supor que precisamos que os testes da classe `livrospock.capitulo08.ImcApiPostTest` do capítulo anterior levem, cada um, no máximo 3 segundos para concluir sua execução. Basta usar a `@Timeout` sobre a classe, como no código a seguir:

```
package livrospock.capitulo08

import spock.lang.Timeout

@Timeout(3)
class ImcApiPostTest extends Specification {
...
}
```

O valor inteiro usado na anotação `@Timeout` é o tempo em **segundos** que **cada um de seus métodos** deve levar para ser executado sem ser marcado como teste com falha.

Caso precisemos determinar o tempo limite da execução de apenas um método de teste, podemos usar essa anotação sobre um teste, como no código de exemplo a seguir:

```
class ImcApiPostTest extends Specification {

    @Timeout(4)
    def 'deve calcular e registrar IMC'() {
     ...
    }
}
```

Nesse exemplo, o teste do método `'deve calcular e registrar IMC'()` tem um tempo de tolerância de 4 segundos para execução.

Como é a indicação do erro em caso de tempo excedido?

Caso o tempo configurado na classe ou no método tenha sido excedido durante a execução do método, o log de erro será como o do exemplo a seguir:

```
<<< FAILURE! - in livrospock.capitulo08.ImcApiPostTest

deveria calcular e registrar IMC(livrospock.capitulo08.ImcApiPostTest)
Time elapsed: 4.002 sec  <<< FAILURE!
org.spockframework.runtime.SpockTimeoutError: Method timed out after 4,00
seconds

        at livrospock.capitulo08.ImcApiPostTest.deveria calcular e
registrar IMC(ImcApiPostTest.groovy:13)
```

E se eu quiser indicar milissegundos, minutos, horas ou outra unidade de tempo? Divido e multiplico?

Não. A anotação `@Timeout` permite indicar a **unidade de tempo** para facilitar a leitura do código do teste e evitar gastar tempo em cálculos de conversão de tempo. Basta usar o atributo `value` de forma explícita e o atributo `unit`, que espera um valor do tipo *enum* `java.util.concurrent.TimeUnit`.

Por exemplo, vamos reconfigurar o último teste para que tenha uma tolerância de tempo de 2 minutos:

```
class ImcApiPostTest extends Specification {

    @Timeout(value=2, unit=TimeUnit.MINUTES)
    def 'deve calcular e registrar IMC'() {
     ...
    }
}
```

Os valores disponíveis na *enum* `TimeUnit` são:

- **NANOSECONDS** (Nanossegundos);
- **MICROSECONDS** (Microssegundos);

- **MILLISECONDS** (Milissegundos);
- **SECONDS** (Segundos);
- **MINUTES** (Minutos);
- **HOURS** (Horas);
- **DAYS** (Dias).

9.3 Executando os testes na ordem em que estão na classe com @Stepwise

Em certos tipos de teste, principalmente em testes funcionais, a sequência dos testes é muito relevante. Lembra dos testes que criamos no *capítulo 8*, no qual testamos uma RESTful API? Imagine que quiséssemos criar uma sequência de testes em que criaríamos um registro de IMC, verificaríamos se ele realmente fora criado, o excluiríamos e verificaríamos se ele realmente fora excluído. Uma alternativa seria criar um único teste, mas ele seria um método muito grande e de baixa coesão, não concorda? E se pudéssemos realizar essa sequência sendo que cada passo estaria em seu próprio método teste? Com o Spock isso é possível pelo simples uso da anotação `@spock.lang.Stepwise`.

A anotação `@Stepwise` **só pode ser usada sobre a definição da classe**. Quando está presente, os métodos de teste da classe são executados na ordem em que estão no arquivo.

Para exemplificar seu uso, vamos criar uma nova classe de testes, a `livrospock.capitulo09.IMCSequenciaCRUDTest`, na qual criaremos os seguintes testes:

- Calcular e registrar IMC;
- Recuperar o IMC recém-criado;
- Excluir registro de IMC recém-criado;
- Tentar recuperar o IMC recém-excluído e não o encontrar.

Note que a ordem correta de execução desses testes é imperativa para que a classe de teste passe. Já pensou se testarmos a exclusão de um registro que ainda nem existe? Segue o código da nova classe de teste:

```
package livrospock.capitulo09

import spock.lang.Stepwise

@Stepwise
class IMCSequenciaCRUDTest extends Specification {
    def 'calcular e registrar IMC'() { ... }

    def 'recuperar o IMC recém-criado'() { ... }

    def 'excluir registro de IMC recém-criado'() { ... }

    def 'tentar recuperar o IMC recém-excluído e não encontrá-lo'() { ...
}
}
```

No código anterior, a exata ordem na qual estão os métodos será a ordem de sua execução devido à presença da anotação `@Stepwise` sobre a classe.

E se houver métodos com `@Ignore`, `@IgnoreIf` ou `@Requires`?

A combinação da `@Stepwise` com as anotações que podem fazer com que testes sejam ignorados não influencia na ordem. Os métodos que tiverem que ser ignorados o serão. Os demais continuam sendo executados na ordem em que estão na classe.

9.4 Automatizando a repetição de testes com @Retry

Em testes de integração ou de sistema, pode ocorrer que um teste falhe devido a questões externas, como indisponibilidade de um outro sistema, por exemplo. Nesses casos, uma solução interessante pode ser programar o teste para que, caso falhe, repita sua execução de forma automática. No Spock, isso é possível com a anotação `@spock.lang.Retry` (disponível a partir do Spock 1.2).

Essa anotação pode ficar sobre a classe ou um método de teste. No primeiro caso, ela define a política de repetição automática em caso de falhas para todos os testes da classe. No segundo, apenas para o teste do método anotado.

A seguir, um exemplo de uso para automatizar a repetição em caso de teste para apenas um método:

```
package livrospock.capitulo09

import spock.lang.Retry

class IMCSequenciaCRUDTest extends Specification {

    @Retry
    def 'excluir registro de IMC recém-criado'() { ... }

}
```

No teste do último código, caso ele não passe, a anotação `@Retry` fará com que o Spock tente novamente sua execução por até **3** vezes, que é quantidade padrão de novas tentativas dessa anotação. Para saber como aumentar ou reduzir esse valor e as demais configurações possíveis dessa anotação, vejamos seus atributos a seguir:

- **`count`** : define a quantidade de novas tentativas após uma falha no teste. *Padrão*: `3` ;
- **`delay`** : define o intervalo de tempo, em milissegundos, entre as novas tentativas. *Padrão*: `0` ;

- **`condition`:** define uma condição a partir da falha para determinar se deve ocorrer uma nova tentativa. Exemplo: só queremos que ocorram novas tentativas caso a mensagem de erro na falha contenha o termo *código não encontrado*. `condition = { failure.message.contains('código não encontrado') }`. *Padrão: não há*;
- **`exceptions`:** define um vetor de classes de exceção. Se pelo menos uma das exceções do vetor for a causa da falha do teste, ocorre uma nova tentativa. Exemplo: queremos que `IOException` provoque novas tentativas. `exceptions = [IOException]`. *Padrão: não há*;
- **`mode`:** define como cada nova tentativa será executada. Só é considerado quando a `@Retry` está sobre a classe. Os valores são da enum `spock.lang.Retry.Mode`, que são: `ITERATION` (repete a execução apenas do teste que falhou); `FEATURE` (repete a execução de todos os testes); e `SETUP_FEATURE_CLEANUP` (além de repetir o teste, repete seus *setup* e *cleanup*, se houverem). *Padrão*: `ITERATION`.

9.5 Compartilhando objetos entre testes com @Shared

Em certos tipos de testes, como nos testes funcionais, a saída de uma etapa pode servir como entrada para a seguinte. Exemplos desse tipo de situação:

- Teste de autenticação como este:
    1. Realize uma autenticação;
    2. Verifique se o usuário da sessão é o que se autenticou anteriormente.
- Teste em uma REST API como este:

1. Crie um recurso;
2. Guarde o identificador que o novo recurso ganhou;
3. Exclua o recurso recém-criado a partir do identificador recém-obtido.

Para situações como essas, o Spock possui a anotação `@spock.lang.Shared`, que **só pode ser usada na definição de atributos de instância**. Ela permite que um atributo anotado com ela seja **compartilhado** entre os diversos métodos de teste.

Quando um atributo em uma classe de testes Spock é anotado com `@Shared`, seu valor é mantido entre os diversos métodos de teste, a não ser que um método o altere. Nesse caso continua o mesmo até que o um próximo método venha a alterá-lo. Esse comportamento está ilustrado na figura a seguir:

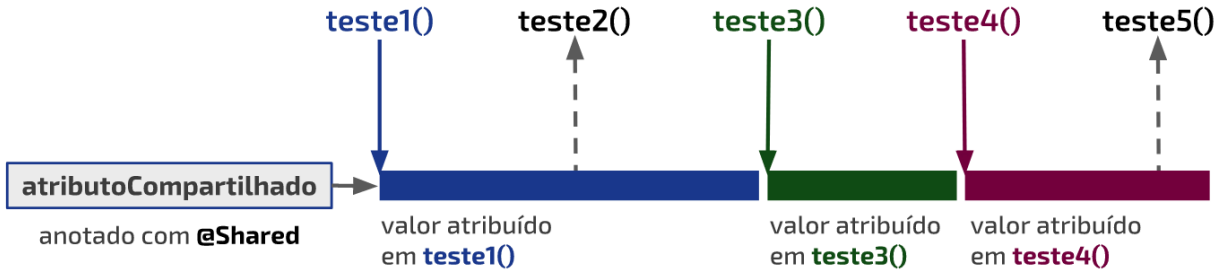


Figura 9.1: Ciclo de vida de um atributo anotado com @Shared

Tomando como exemplo a classe de teste `IMCSequenciaCRUDTest` do tópico anterior, poderíamos definir um atributo compartilhado entre os testes, como no código-fonte a seguir:

```
// pacote e imports

@Stepwise
class IMCSequenciaCRUDTest extends Specification {

    @Shared
    def idRecursoCriado

    def 'calcular e registrar IMC'() {
```

```
        // blocos do teste
        idRecursoCriado = resposta.id
    }

    def 'recuperar o IMC recém-criado'() {
        // blocos do teste usando o idRecursoCriado
    }

    def 'excluir registro de IMC recém-criado'() {
        // blocos do teste usando o idRecursoCriado
    }
}
```

Nesse último código de exemplo, o atributo `idRecursoCriado` foi alterado no teste `'calcular e registrar IMC'()`. Como a classe de testes está anotada com `@Stepwise`, os métodos serão executados na sequência em que estão no arquivo, portanto o valor de `idRecursoCriado` será atribuído no primeiro método e permanecerá o mesmo para os demais.

9.6 Restaurando propriedades do sistema com @RestoreSystemProperties

Alguns cenários de teste exigem que sejam configuradas algumas propriedades do sistema. Porém, para evitar efeitos colaterais em outros testes do mesmo projeto, é mais seguro reverter quaisquer alterações nessas propriedades após a execução dos testes. Para essa tarefa, podemos usar a anotação `@spock.util.environment.RestoreSystemProperties` sobre a classe de teste.

Por exemplo, vamos supor que tivemos que alterar uma propriedade na classe do capítulo anterior `livrospock.capitulo08.ImcApiPostTest`, mas queremos que essa alteração seja revertida após todos os testes dessa classe. Basta fazer como no código a seguir:

```
@RestoreSystemProperties
class ImcApiPostTest extends Specification {

    def 'deve calcular e registrar IMC'() {
        given:
        System.setProperty("api-token", "a5e61b00-db1f-406b")
        ...
    }

    // demais testes da classe
```

Após a execução de todos os testes da classe do último código, a propriedade de sistema `api-token` será excluída, caso tenha sido criada no teste, ou voltará para seu valor anterior, caso já existisse antes da execução dos testes dessa classe.

9.7 Dando um título para a classe de testes com @Title

Não é possível que o nome de uma classe Groovy seja uma frase, como podemos fazer com os métodos. Assim, ficamos presos aos nomes no estilo *PascalCase*. Porém, podemos descrever o objetivo macro de uma classe de testes por meio da anotação `spock.lang.Title`. Vejamos como fazer isso no próximo código de exemplo:

```
package livrospock.capitulo09

import spock.lang.Title

@Title("Testes de cenários de faixas de IMC")
class FaixaImcTest extends Specification {
    // métodos de teste
}
```

Vimos que o valor da anotação `@Title` deve conter uma explicação sobre o objetivo da classe de testes. A princípio, esse texto não aparece em lugar algum no processo de execução. Os logs do Maven/Gradle ignoram essa anotação. Sua relevância se dá porque é melhor uma explicação em uma anotação, o que é algo explícito e formal, do que em um comentário de código.

Outro motivo relevante em se usar essa técnica é a ferramenta Spock Reports Extension, que veremos no capítulo 10, pois ela faz uso dos conteúdos das `@Title` .

9.8 Indicando referências

Em certos projetos, é pertinente indicar referências teóricas para os cenários dos testes e/ou indicar que *issue* motivou a criação do teste. Por exemplo, em projetos com cálculos de impostos (onde seriam indicadas as leis) e projetos de integrações de sistemas (onde seriam indicadas as origens dos sistemas externos). Para esses tipos de referências, há anotações do Spock que as deixam mais explícitas e claras.

Indicando referências com `@See`

Imagine que determinado teste foi criado para garantir que uma funcionalidade siga uma norma do governo, uma verdade científica comprovada ou uma regra específica do cliente segundo algum documento oficial dele disponível na internet. Você pode simplesmente criar um `javadoc` para a classe e/ou métodos de teste. Ou, se quiser um recurso mais assertivo e simples de ler, pode usar a anotação `@spock.lang.See` do Spock.

A anotação `@See` espera como valor uma ou várias URLs, indicando a fonte on-line que contém o embasamento ou a justificativa do teste.

Essa anotação pode estar tanto na definição da classe quanto na dos métodos de teste.

Como exemplo, vamos supor que quiséssemos deixar claro onde nos baseamos para montar o cenário de testes da calculadora de IMC do *capítulo 5*. Poderíamos fazer a seguinte alteração na classe `livrospock.capitulo05.FaixaImcTest`:

```
package livrospock.capitulo05

import spock.lang.See

class FaixaImcTest extends Specification {

    @See('http://bvsms.saude.gov.br/bvs/dicas/215_obesidade.html')
    def 'IMC deve estar na faixa correta'() {
     ...
    }
```

Veja que fica muito claro que temos uma referência do tipo *"veja mais em"* sobre o método `'IMC deve estar na faixa correta'()`. Se fosse necessário usar mais de uma referência, sem problemas, basta usar um vetor de URLs na `@See`, como mostra o próximo código:

```
class FaixaImcTest extends Specification {

    @See(['http://bvsms.saude.gov.br/bvs/dicas/215_obesidade.html',
'http://intranet.sesporte.ce.gov.br/imc/'])
    def 'IMC deve estar na faixa correta'() {
     ...
    }
```

Indicando referências com `@Issue`

Imagine que determinado teste foi criado para cumprir algo que foi solicitado e está registrado em uma *issue* do GitHub, BitBicker, Jira ou afins. Se quiser um recurso mais assertivo e simples de ler que o `javadoc`, pode usar a anotação `@spock.lang.Issue` do Spock.

A anotação `@Issue` espera como valor uma ou várias URLs, indicando o endereço da *issue*. Essa anotação pode estar tanto na definição da classe quanto na dos métodos de teste.

Como exemplo, vamos supor que quiséssemos deixar claro em qual *issue* nos baseamos para montar o cenário de testes da calculadora de IMC do *capítulo 5*. Poderíamos fazer a seguinte alteração na classe `livrospock.capitulo05.FaixaImcTest` :

```
import spock.lang.Issue

class FaixaImcTest extends Specification {

    @Issue('https://github.com/perfil/repositorio/issues/12')
    def 'IMC deve estar na faixa correta'() {
     ...
    }
```

Veja que fica muito claro que temos uma referência do tipo *"detalhes sobre esse teste na issue"* sobre o método `'IMC deve estar na faixa correta'()` . Se fosse necessário referenciar mais de uma *issue*, bastaria usar um vetor de URLs na `@Issue` , como no próximo código:

```
class FaixaImcTest extends Specification {

    @Issue(['https://github.com/perfil/repositorio/issues/12',
'https://github.com/perfil/repositorio/issues/144'])
    def 'IMC deve estar na faixa correta'() {
     ...
    }
```

Os conteúdos de `@See` e `@Issue` também são usados pela ferramenta que veremos no próximo capítulo.

Conclusão

Neste capítulo, vimos recursos avançados do Spock framework que propõem soluções prontas simples para pequenas necessidades que possam surgir em nossos códigos de testes, que vão desde a capacidade de ignorar testes, definir um limite de tempo para execução até documentar melhor os testes no próprio código. No próximo capítulo, veremos como configurar a geração de **relatórios profissionais em HTML das execuções dos testes** com Spock.

Capítulo 10
Relatórios profissionais com Spock Reports Extension

Agora que já vimos como criar testes dos mais diversos tipos e com diferentes recursos e opções, vamos ver como é possível gerar um relatório profissional com o **Spock Reports Extension**, que aproveita recursos estudados nos capítulos anteriores.

Por que relatórios se a IDE e o Maven/Gradle exibem detalhes dos testes?

As saídas de testes padrão do Maven/Gradle são arquivos de *log* em formato de texto simples. Para uma meia dúzia de testes isso pode até ajudar, mas em projetos com centenas ou milhares de testes passa a ser um artefato grande e de difícil leitura.

Já o Spock Reports Extension gera um relatório navegável que é um conjunto de páginas HTML. Assim, mesmo um conjunto de milhares de testes fica em um formato mais organizado e amigável.

10.1 Configurando o Spock Reports Extension

Essa ferramenta não faz parte do framework Spock, porém existe desde 2013 e está em constante evolução, tendo 3 versões lançadas em 2017, 3 em 2018 e, até julho de 2019, mais 1 versão. Para usá-la em um projeto com Spock, basta adicionar sua dependência no escopo de testes, como mostram os códigos a seguir:

Dependência Maven

```
<dependency>
    <groupId>com.athaydes</groupId>
```

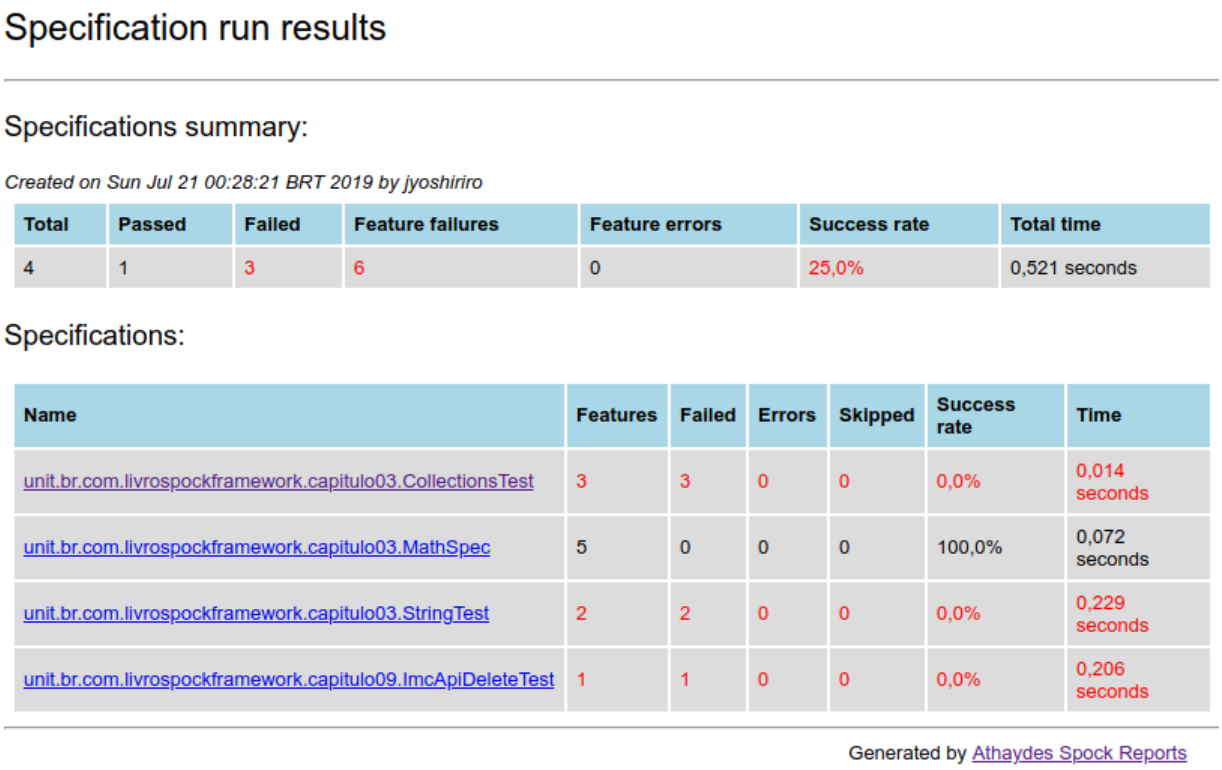
```
    <artifactId>spock-reports</artifactId>
    <version>2.5.1-groovy-4.0</version>
    <scope>test</scope>
</dependency>
```

Dependência Gradle

```
testImplementation 'com.athaydes:spock-reports:2.5.1-groovy-4.0'
```

Em ambos, usamos a versão `2.5.1-groovy-4.0`, que era a mais atual até o fechamento desta obra.

Uma vez que sua dependência está no projeto, sempre que um teste em Spock for executado, seja de forma individual pela IDE seja pelo Maven/Gradle, serão criados e/ou atualizados arquivos HTML no diretório `build/spock-reports` (`build` fica na raiz do projeto). Nesse diretório, sempre haverá um arquivo `index.html`, cuja aparência é como a da figura a seguir:



Specification run results

Specifications summary:

Created on Sun Jul 21 00:28:21 BRT 2019 by jyoshiriro

Total	Passed	Failed	Feature failures	Feature errors	Success rate	Total time
4	1	3	6	0	25,0%	0,521 seconds

Specifications:

Name	Features	Failed	Errors	Skipped	Success rate	Time
unit.br.com.livrospockframework.capitulo03.CollectionsTest	3	3	0	0	0,0%	0,014 seconds
unit.br.com.livrospockframework.capitulo03.MathSpec	5	0	0	0	100,0%	0,072 seconds
unit.br.com.livrospockframework.capitulo03.StringTest	2	2	0	0	0,0%	0,229 seconds
unit.br.com.livrospockframework.capitulo09.ImcApiDeleteTest	1	1	0	0	0,0%	0,206 seconds

Generated by Athaydes Spock Reports

Figura 10.1: Exemplo de página inicial de um relatório gerado com Spock Reports

A primeira informação é um resumo rápido dos testes: quantos foram executados, quantos passaram, quantos não passaram, quantos deram erro, taxa de sucesso e tempo total de execução. Em seguida, temos as mesmas informações (com o adicional de contagem de testes ignorados) para cada especificação (teste) executada. É criada uma página HTML por classe de teste executada. Assim, ao clicar no nome de alguma classe de testes, somos direcionados para sua respectiva página, como a da tela na próxima figura:

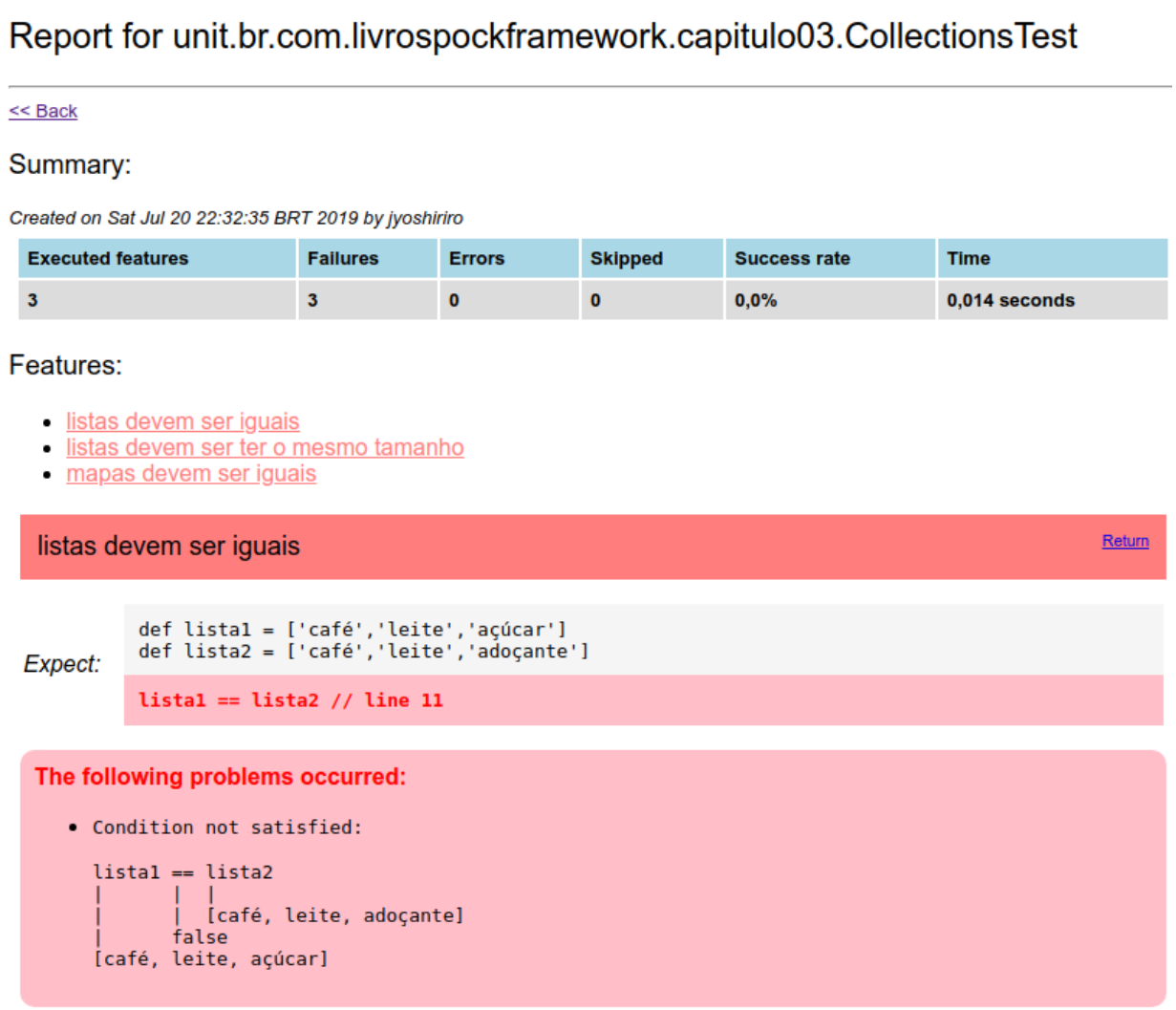


Figura 10.2: Exemplo de página de detalhe de classe de testes gerada com Spock Reports

Na página de cada classe de testes, são exibidos vários detalhes da execução dos seus testes. E, no caso de testes que falharam, o

detalhe da falha é exibido de forma bem clara e com interessante destaque visual.

10.2 Alterando o diretório de geração do relatório

Talvez o diretório `build/spock-reports` vá de encontro a algum padrão dos projetos de sua empresa ou simplesmente você não goste do padrão do Spock Reports Extension. Sem problemas, é muito simples alterar esse diretório. Basta criar um arquivo chamado `SpockConfig.groovy` no seu `src/main/resources` e escrever o seguinte conteúdo nele:

```
spockReports {
    set 'com.athaydes.spockframework.report.outputDir': 'relatorio-testes'
}
```

O código de exemplo faria com que os arquivos HTML dos testes fossem gerados no diretório `relatorio-testes` na raiz do projeto.

10.3 Anotações @Title, @See e @Issue são usadas nos relatórios

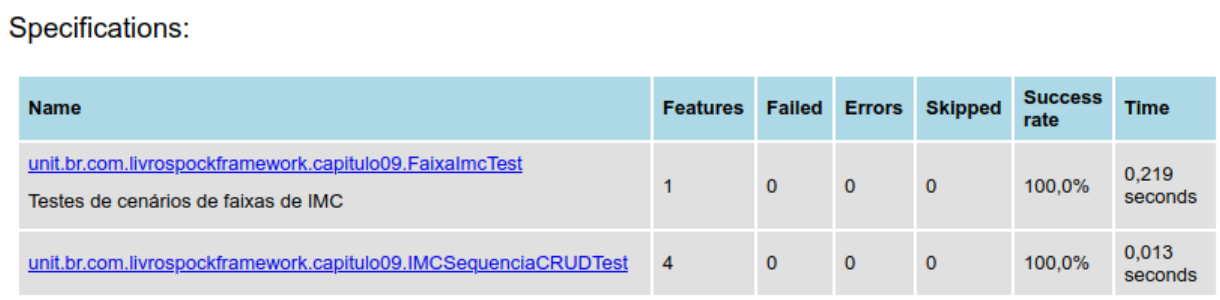
No *capítulo 9* estudamos as anotações `@Title`, `@See` e `@Issue`. Nos logs padrão do Maven/Gradle, os conteúdos dessas anotações são ignorados, porém o Spock Reports Extension os aproveita para detalhar melhor os testes executados.

Por exemplo, considere que a classe `FaixaImcTest` está anotada com `@Title` como no código a seguir:

```
@Title("Testes de cenários de faixas de IMC")
class FaixaImcTest extends Specification {

}
```

Nesse caso, seu conteúdo fica visível abaixo do nome da classe da página inicial do relatório de testes, como é possível ver na figura a seguir:


Specifications:

Name	Features	Failed	Errors	Skipped	Success rate	Time
unit.br.com.livrospockframework.capitulo09.FaixaImcTest Testes de cenários de faixas de IMC	1	0	0	0	100,0%	0,219 seconds
unit.br.com.livrospockframework.capitulo09.IMCSequenciaCRUDTest	4	0	0	0	100,0%	0,013 seconds

Figura 10.3: Exemplo de página inicial de um relatório gerado com Spock Reports

Vamos considerar que essa classe possui um método anotado com `@See` e `@Issue`, como no próximo código:

```
@Title("Testes de cenários de faixas de IMC")
class FaixaImcTest extends Specification {

    @See(['http://bvsms.saude.gov.br/bvs/dicas/215_obesidade.html',
'http://intranet.sesporte.ce.gov.br/imc/'])
    @Issue('https://github.com/perfil/repositorio/issues/12')
    def 'IMC deve estar na faixa correta'() {

    }

}
```

Se formos para a página de detalhes da `FaixaImcTest`, vemos novamente o conteúdo da `@Title` além dos conteúdos das `@See` e `@Issue`, como podemos notar na próxima figura:

Figura 10.4: Exemplo de página de detalhe de classe de testes gerada com Spock Reports

MADE IN BRAZIL

O criador dessa ferramenta é **Renato Athaydes** e ele é brasileiro! Ele lançou a primeira versão dela em 2013 e vem atualizando ela constantemente. A versão usada aqui é de agosto de 2023.

O código do projeto no GitHub está disponível em: https://github.com/renatoathaydes/spock-reports.

O perfil do Renato Athaydes no LinkedIn é: https://www.linkedin.com/in/renatoathaydes.

Conclusão

Neste capítulo, vimos como gerar um relatório profissional em HTML de forma simples com o Spock Reports Extension. No próximo capítulo, veremos como **testar metaprogramação** com Spock.

CAPÍTULO 11
Testando metaprogramação (anotações)

Já estudamos até aqui a imensa maioria dos testes possíveis de serem criados com Spock, e várias vezes lembramos a importância dos testes automatizados. Porém, há um certo tipo de detalhe que programamos e que costuma ser negligenciado nos testes: a **metaprogramação**. Neste capítulo, veremos como testar em projetos Java a metaprogramação, ou seja, anotações.

11.1 Metaprogramação em Java: anotações

Na plataforma Java, metaprogramação pode ser feita por meio de *annotations* (comumente traduzido como **anotações**). Esse recurso foi introduzido na versão 5 da linguagem Java (lançada em 2004). Desde então, passou a ser cada vez mais usado tanto pelas mais diversas especificações oficiais da plataforma (por exemplo: `JPA`, `Servlets API`) quanto pelos inúmeros frameworks e bibliotecas compatíveis com a JVM (como soluções `Spring` e `GSON`).

METAPROGRAMAÇÃO

Segundo o livro *Generative Programming: Methods, Tools, and Applications*, escrito por Krzysztof Czarnecki e Ulrich Eisenecker (2000), metaprogramação é a técnica de programação na qual algumas partes de um programa funcionam como configurações. Exemplos disso são pré-processadores e compiladores, ou quando um programa manipula outro como se fosse um repositório de dados.

No livro da Casa do Código *Componentes reutilizáveis em Java com Reflexão e Anotações*, escrito por Eduardo Guerra (2014), metaprogramação é descrita como um código que trabalha com o próprio código. Com ela, um programa pode realizar ações computacionais a respeito dele mesmo.

Em Java, anotações podem estar presentes em diferentes partes do código, tais como:

- **Na definição da classe**. Nesse caso, definimos uma metaprogramação que influenciará toda a classe. Neste livro, há exemplos de uso desse tipo de anotação no *capítulo 7* (`@Entity`) e no *capítulo 9* (`@RestoreSystemProperties` , `@Stepwise` e outras).
- **Na definição de um atributo**. Aqui, definimos uma metaprogramação que influenciará somente o atributo anotado. Neste livro, há exemplos de uso desse tipo de anotação no *capítulo 7* (`@Column`).
- **Na definição de um método**. Nesse caso, definimos uma metaprogramação que influenciará somente o método anotado. Neste livro, há exemplos de uso desse tipo de anotação no *capítulo 5* (`@Unroll`) e no *capítulo 9* (`@Ignore` , `@Timeout` e outras).

11.2 Por que testar anotações?

As consequências de retirar ou alterar os atributos de uma anotação muitas vezes passam despercebidas nos testes unitários e de integração. Isso porque as configurações de algumas anotações só são solicitadas durante a execução de funcionalidades pontuais de um programa. Um exemplo disso são algumas anotações da especificação `JPA`: apesar de existir uma configuração que solicita ao provedor `JPA` que valide o banco de dados conforme o mapeamento do projeto, ainda podem ocorrer problemas em tempo de execução devido a alterações em anotações.

Um exemplo concreto seria caso invertêssemos acidentalmente o atributo `nullable` de uma anotação `@Column`. A etapa de validação do `JPA` **não vai notar essa diferença**, tampouco vai alterar a definição do campo na tabela do banco. Então, caso não exista algum teste que simule a criação ou alteração de registro da tabela que possui um campo que sofreu essa mudança, o sistema vai lançar uma exceção durante uma dessas operações.

Nesse caso, poderíamos criar um cenário de teste para lidar com a obrigatoriedade do campo em questão. Só que **não seria um teste simples**, pois anotações não costumam ser lidas por mecanismos simples. Para se ter uma ideia, seria necessário um teste de integração que envolvesse várias classes, configurações e *mocks* só para testar cenários para um campo obrigatório de uma entidade `JPA`.

Para situações assim, já que usamos anotações para facilitar configurações por meio de metaprogramação, fica mais fácil também usar metaprogramação para testar essas configurações. A seguir, vamos usar uma entidade `JPA` para demonstrar como realizar esse tipo de teste com Spock.

Além da especificação `JPA`, há inúmeras tecnologias para a plataforma Java nas quais anotações são muito usadas e podem ser mais bem cobertas por testes. Alguns exemplos:

- Spring / Spring Boot;
- Especificação JEE (CDI, EJB, JAX-RS etc.);
- Jackson;
- Gson.

11.3 Testando anotações de uma entidade JPA

Conforme dito há pouco, há várias tecnologias para Java que usam bastante do recurso de anotações. Consequentemente, torna-se prudente uma maior cobertura dessas metaprogramações. Neste capítulo, vamos usar como estudo de caso os testes de anotações da `JPA`. Vamos supor que temos uma entidade que está corretamente mapeada para sua tabela no banco de dados. É a entidade `Lutador`, mapeada para a tabela `lutador` e cujo código é:

```
package br.com.livrospockframework.capitulo11.entity;

import javax.persistence.*;

@Entity
public class Lutador {

    @Id
    @GeneratedValue(strategy=GenerationType.IDENTITY)
    @Column(name="id_lutador")
    private Long id;

    @Column(nullable = false, length=30)
    private String nome;

    @Column(scale=2)
    private double peso;

    private Calendar nascimento;

    // getters e setters
```

Queremos garantir que ela permaneça corretamente mapeada, então criamos um teste unitário para testar suas anotações `JPA`. É a classe `LutadorTest`, cujo início é como o do código a seguir:

```
package livrospock.capitulo11;

import java.lang.reflect.Field
import javax.persistence.*
import br.com.livrospockframework.capitulo11.entity.Lutador

class LutadorTest extends Specification {

    Class classe = Lutador.class;

    // testes
}
```

Definimos o atributo `classe` apenas para simplificar o código nos testes da classe. A seguir, os cenários de testes de metaprogramação para cada uma das anotações `JPA` da classe.

Testando a anotação da classe

Para testar a metaprogramação da classe, criamos o teste a seguir:

```
class LutadorTest extends Specification {

    def 'classe deve estar anotada com @Entity vazia e sem @Table'() {
        expect:
        !classe.getAnnotation(Entity).name()
        !classe.getAnnotation(Table)
    }

    // outros testes
```

Nesse último teste, apenas criamos um bloco `expect` para testar primeiro se existe a anotação `@Entity` na classe e se seu atributo

`name` é vazio ou nulo. Caso a anotação `@Entity` não esteja presente, ocorrerá uma `NullPointerException` ao tentar invocar o `name()` e o teste falhará corretamente. Em seguida, testamos se a classe **não está anotada** com `@Table` . Essa é uma anotação comum em projetos `JPA` , mas, como não foi usada, espera-se que não esteja presente na entidade `Lutador` . O fato de ela passar a existir pode gerar problemas em tempo de execução, como a busca por uma tabela que não existe, por exemplo.

Caso o teste falhe pela ausência na anotação `@Entity` , o log de erro será como o do exemplo a seguir:

```
Failed tests:
  LutadorTest.classe deve estar anotada com @Entity vazia e sem @Table:21
Condition failed with Exception:

!classe.getAnnotation(Entity).name()
|      |                     |
|      null                  java.lang.NullPointerException: Cannot invoke
method name() on null object
class br.com.livrospockframework.capitulo10.entity.Lutador
```

Caso o teste falhe pela presença do atributo `name` na anotação `@Entity` , o log de erro será como o do exemplo a seguir:

```
Failed tests:
  LutadorTest.classe deve estar anotada com @Entity vazia e sem @Table:21
Condition not satisfied:

!classe.getAnnotation(Entity).name()
||      |                     |
||      |                     ehlaiah
||      @javax.persistence.Entity(name=ehlaiah)
|class br.com.livrospockframework.capitulo10.entity.Lutador
false
```

Caso o teste falhe pela presença da anotação `@Table` , o log de erro será como o do exemplo a seguir:

```
Failed tests:
  LutadorTest.classe deve estar anotada com @Entity vazia e sem @Table:22
Condition not satisfied:

!classe.getAnnotation(Table)
||      |
||      @javax.persistence.Table(schema=, name=, uniqueConstraints=[],
indexes=[], catalog=)
|class br.com.livrospockframework.capitulo10.entity.Lutador
false
```

Testando as anotações do atributo `id`

Para testar a metaprogramação do atributo `id`, criamos o teste a seguir:

```
class LutadorTest extends Specification {

    def '"id" deve estar anotado com @Id, @GeneratedValue
strategy=IDENTITY e @Column name="id_lutador" '() {
        setup:
        def campo = classe.getDeclaredField('id')
        def anotacaoId = campo.getDeclaredAnnotation(Id)
        def anotacaoGeneratedValue =
campo.getDeclaredAnnotation(GeneratedValue)
        def anotacaoColumn = campo.getDeclaredAnnotation(Column)

        expect:
        anotacaoId
        anotacaoGeneratedValue.strategy() == GenerationType.IDENTITY
        anotacaoColumn.name() == 'id_lutador'
    }

    // outros testes
```

No último teste, criamos um bloco `setup` para recuperar as anotações que queremos verificar junto ao atributo `id` (`@Id`, `@GeneratedValue` e `@Column`). No bloco `expect`, verificamos primeiro se foi encontrada a `@Id`. Depois, testamos se a `@GeneratedValue` foi

encontrada e se o seu atributo `strategy` é `GenerationType.IDENTITY`. Por fim, verificamos se a `@Column` foi encontrada e se o seu atributo `name` é `id_lutador`.

Testando as anotações do atributo `nome`

Para testar a metaprogramação do atributo `nome`, criamos o teste a seguir:

```
class LutadorTest extends Specification {

    def '"nome" deve estar anotado com @Column nullable = false, length=30
'() {
        setup:
        def anotacaoColumn =
classe.getDeclaredField('nome').getDeclaredAnnotation(Column)

        expect:
        !anotacaoColumn.nullable()
        anotacaoColumn.length() == 30
    }

    // outros testes
```

No último teste, criamos um bloco `setup` para recuperar a anotação que queremos verificar junto ao atributo `nome` (`@Column`). No bloco `expect`, verificamos primeiro se ela foi encontrada e se seu atributo `nullable` é `false`. Em seguida, verificamos se o seu atributo `length` é `30`.

Testando as anotações do atributo `peso`

Para testar a metaprogramação do atributo `peso`, criamos o teste a seguir:

```
class LutadorTest extends Specification {

    def '"peso" deve estar anotado com @Column nullable = false, scale=2
'() {
```

```
        setup:
        def anotacaoColumn =
classe.getDeclaredField('peso').getDeclaredAnnotation(Column)

        expect:
        anotacaoColumn.scale() == 2
    }

    // outros testes
```

No último teste, criamos um bloco `setup` para recuperar a anotação que queremos verificar junto ao atributo `peso` (`@Column`). No bloco `expect` , verificamos se ela foi encontrada e se seu atributo `scale` é `2` .

Testando as anotações do atributo `nascimento`

Para testar a metaprogramação do atributo `nascimento` , criamos o teste a seguir:

```
class LutadorTest extends Specification {

    def '"nascimento" deve estar SEM anotações '() {
        expect:
        !classe.getDeclaredField('nascimento').getDeclaredAnnotations()
    }

    // outros testes
```

No último teste, criamos um bloco `expect` apenas para testar se o atributo `nascimento` não possui anotações.

Conclusão

Neste capítulo, vimos como criar testes unitários para metaprogramação em Java. Com esse tipo de teste, evitaremos erros que só poderiam ser percebidos em tempo de execução (ou pior, em produção). Também evitamos a criação de complexos testes de integração apenas para validar requisitos programados via

anotações. Com os exemplos que demonstram a facilidade em realizar esse tipo de teste, esperamos estimular você a praticá-lo com maior frequência e qualidade. No próximo capítulo, veremos como realizar **testes de integração automatizados em projetos Spring**.

Capítulo 12
Testes de integração em projetos Spring

Spring é um conjunto de bibliotecas e frameworks usados na criação dos mais diversos tipos de sistemas na plataforma Java. Segundo pesquisa realizada pela Eclipse Foundation (ECLIPSE, 2024), a adoção dos frameworks Spring é maior que da própria especificação JEE, que é a tecnologia oficial da Oracle para sistemas web e corporativos (veja a próxima figura).

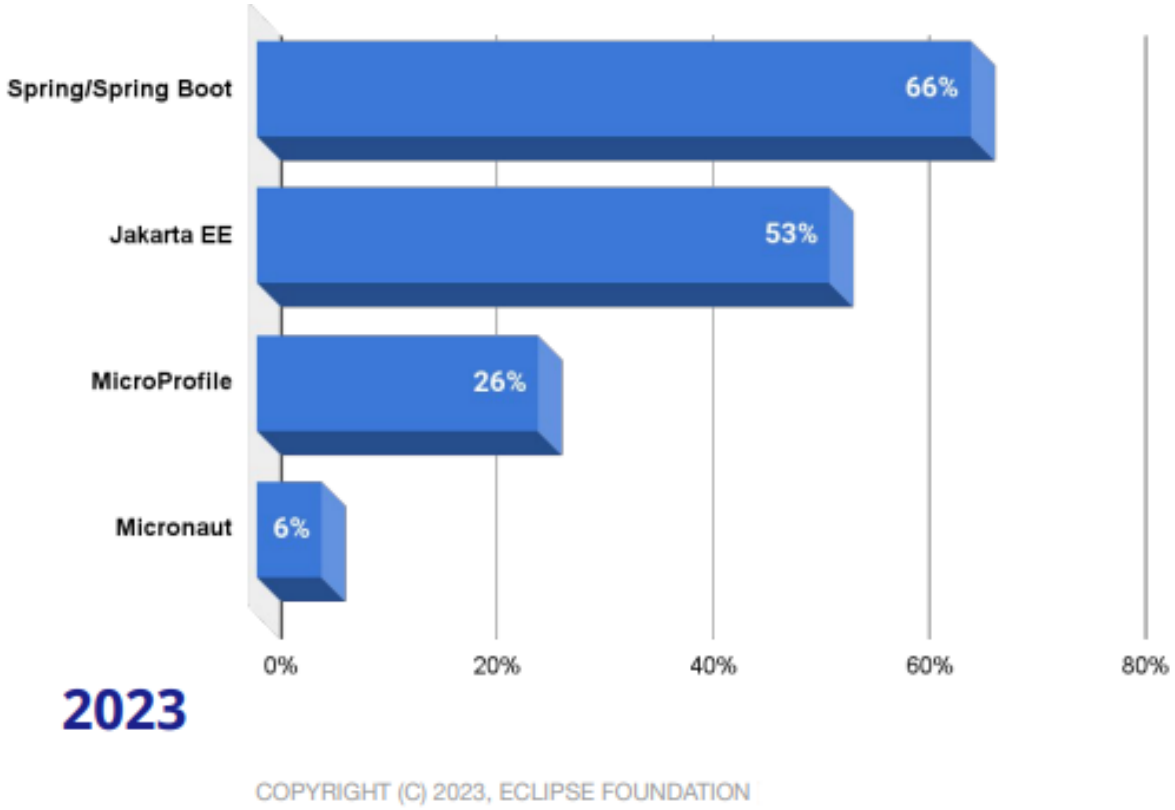


Figura 12.1: Adoção dos frameworks Spring x JEE. Fonte: ECLIPSE, 2024

Neste capítulo, focaremos na criação de **testes de integração**. Isso porque, para criar testes unitários, basta fazer conforme estudamos nos capítulos anteriores desta obra. Já estudamos o

conceito desse tipo de teste no capítulo 1, *Por que usar Spock framework?*, mas que tal relembrarmos?

> **Testes de integração**
>
> Usados para testar a integração entre duas ou mais partes do programa ou a integração de uma parte do programa a um elemento externo (banco de dados, REST API etc.). É um complemento aos testes unitários, pois uma parte que funciona perfeitamente sozinha não terá, necessariamente, um funcionamento perfeito em conjunto com outra(s). Fazendo uma analogia com a fabricação de um carro, seria como testar a integração entre todas as peças que compõem o câmbio de um carro, ou seja, testar o câmbio montado.

Existe uma relevância muito grande em criar testes de integração para essas tecnologias porque elas promovem a integração de vários componentes e abstração de várias funcionalidades nos projetos em que são usadas. Em um projeto Spring Boot, existem componentes e abstrações desde o acesso ao banco de dados até exposição de *endpoints* em uma REST API.

Serão usados dois estudos de caso: um para o Spring Boot e outro para Spring MVC. Criar testes de integração para ambos é feito de forma muito parecida, pois ambos usam várias bibliotecas Spring em comum. O estudo de caso mais completo será o do Boot. No estudo do MVC, será demonstrado como testar apenas algumas de suas funcionalidades específicas.

Importante! Caso use Spring/Spring Boot em seu projeto, não é necessário adicionar as dependências de *mocks* mencionadas no capítulo 7, *Testando com a ajuda de mocks*.

12.1 Spring TestContext framework e Spring Module do Spock

Para facilitar a criação de testes de integração para os frameworks Spring, foi criado um framework de testes chamado **Spring TestContext**. Ele contém uma série de bibliotecas que permite que um teste de integração inicie um contexto completo de uma aplicação Spring, dando a possibilidade de testar todos os componentes de um projeto de forma integrada.

Para fazer uso de todas as funcionalidades do *Spring TestContext* em testes com Spock, existe um módulo do Spock chamado **Spring Module**. Para usá-lo, é preciso adicionar uma dependência ao projeto, que consta no código a seguir:

```
<dependency>
    <groupId>org.spockframework</groupId>
    <artifactId>spock-spring</artifactId>
    <version>2.3-groovy-4.0</version> <!-- ou a versão que preferir -->
    <scope>test</scope>
</dependency>
```

Essa dependência é necessária para os estudos de caso abordados neste capítulo.

O *Spring TestContext framework* é muito grande. O objetivo aqui é abordá-lo de forma bem básica para compreender como ele foi habilitado pelo *Spring Module* do Spock. Caso queira se aprofundar, existe um tutorial completo sobre o *Spring TestContext framework* em sua página no site do Spring, em https://docs.spring.io/spring/docs/current/spring-framework-reference/testing.html.

Existem mais módulos Spock para facilitar a criação de testes de integração para outras tecnologias como *Guice*, *Tapestry*, *Unitils* e *Grails*, mas que não fazem parte desta obra.

12.2 Testes de integração em um projeto Spring Boot

O Spring Boot é um framework muito usado para criar REST APIs, possuindo uma série de componentes que abstraem boa parte do trabalho repetitivo e perigoso envolvido na criação desse tipo de API.

Resumo do Projeto Spring Boot

O estudo de caso aqui será um pequeno projeto de **cadastro de lutadores de MMA**. Esse projeto será constituído por um conjunto básico de classes no estilo **Entity** (JPA) + **Repository** (Spring) + **Service** (Spring) + **Spring REST Controller** (Spring), conforme o diagrama de classes da figura a seguir:

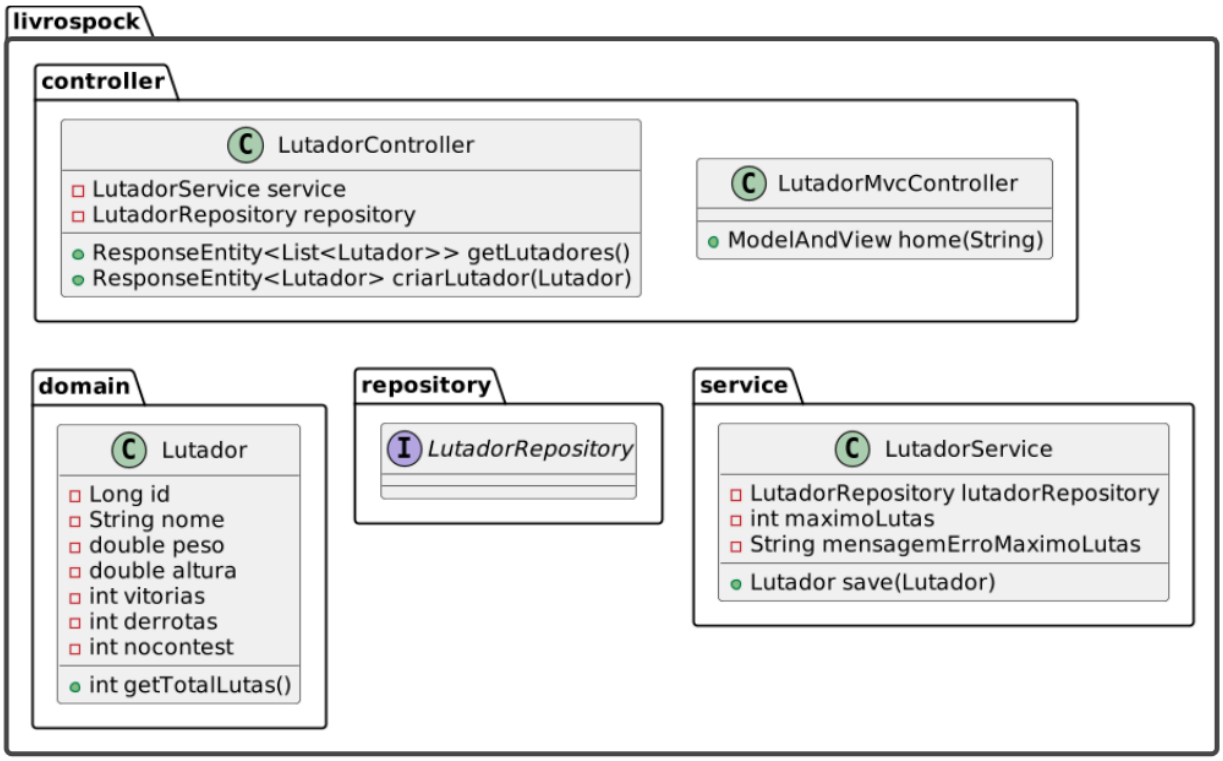


Figura 12.2: Diagrama de classes do projeto de Lutadores de MMA em Spring Boot

A classe `LutadorMvcController` será usada no tópico **Testes de integração em um projeto Spring MVC**.

Classe `Lutador` (*JPA Entity*)

```
// pacote e imports JPA

@Entity
public class Lutador {

    @Id
    @GeneratedValue(strategy=GenerationType.IDENTITY)
    private Long id;

    @Column(nullable=false)
    private String nome;

    private double peso;

    private double altura;

    private int vitorias;

    private int derrotas;

    private int nocontest;

    // construtores, getters e setters

    public int getTotalLutas() {
        return vitorias + nocontest + derrotas;
    }
```

A classe `Lutador` é uma entidade JPA simples, com um `id` (identificador) de valor gerado com autoincremento e alguns atributos simples, sem relacionamento com nenhuma outra entidade. O atributo `nome` é obrigatório. Os demais também, mas, como são de tipos numéricos primitivos, serão inicializados com `0`

(zero) automaticamente. Seu método `getTotalLutas()` retorna a soma de vitórias, derrotas e *no contests*.

Aliás, se você não sabe o que é *no contest*, resumidamente, é quando não há vencedor nem derrotado no combate. Isso pode ocorrer quando a luta termina prematuramente ou quando tem seu resultado alterado posteriormente.

Interface `LutadorRepository` (*Spring Repository*)

```
// pacote e imports Spring

public interface LutadorRepository extends JpaRepository<Lutador, Long> {

}
```

A interface `LutadorRepository` é uma extensão de `JpaRepository`, funcionando como um *Repository* cuja classe de entidade é `Lutador` e a de identificação (chave primária) é `Integer`.

Classe `LutadorService` (*Spring Service*)

```
// pacote e imports Spring e JPA

@Service
public class LutadorService {

    @Autowired
    private LutadorRepository lutadorRepository;

    @Value("${maximo.lutas}")
    private int maximoLutas;

    @Value("${erro.maximo-lutas}")
    private String mensagemErroMaximoLutas;

    public Lutador save(Lutador lutador) {
        if (lutador.getTotalLutas() >  maximoLutas) {
            throw new ResponseStatusException(HttpStatus.BAD_REQUEST,
```

```
mensagemErroMaximoLutas);
        }
        return lutadorRepository.save(lutador);
    }
}
```

A classe `LutadorService` está anotada com `@Service`, tornando-a um componente do tipo *Service* do Spring (ou seja, terá uma única instância em toda a aplicação). Ela possui apenas um atributo, o `lutadorRepository` do tipo `LutadorRepository`, para que possamos realizar operações da JPA nos métodos. Ele será injetado em tempo de execução devido à anotação `@Autowired` do Spring.

Os atributos anotados com `@Value` servem para recuperar propriedades de aplicação que serão usadas no método `save(Lutador lutador)`, explicado a seguir.

Seu método `save(Lutador lutador)` retorna um `Lutador`. Ele primeiro verifica se a quantidade de lutas do `Lutador` não ultrapassa o máximo de lutas por lutador configurado na aplicação (propriedade `maximo.lutas`). Caso ultrapasse, uma exceção do tipo `ResponseStatusException` é lançada com o status *400 Bad Request* e com a mensagem definida na propriedade `erro.maximo-lutas`. Caso o total de lutas esteja certinho, o `Lutador` é salvo no banco.

Classe `LutadorController` (Spring REST Controller)

```
// pacote e imports Spring e JPA

@RestController
@RequestMapping("/lutadores")
public class LutadorController {

    @Autowired
    private LutadorService service;

    @Autowired
    private LutadorRepository repository;
```

```
    // métodos de EndPoints
```

A classe `LutadorController` está anotada com `@RestController`, do Spring. Isso a torna capaz de abrigar *endpoints* da REST API da aplicação Spring Boot.

A anotação `@RequestMapping("/lutadores")` indica que todos os *endpoints* dessa *controller* começam com **`/lutadores`**.

Os atributos `LutadorService service` e `LutadorRepository repository` estão anotados com `@Autowired` para que sejam injetados pelo Spring em tempo de execução, usando a classe e a interface já explicadas neste tópico.

Endpoint *GET /lutadores* da `LutadorController`

```
public class LutadorController {

    // atributos
    // método getLutadorMaisVitorias()

    @GetMapping
    public ResponseEntity getLutadores() {

        return this.repository.count() == 0L
                ? ResponseEntity.noContent().build()
                : ResponseEntity.ok(this.repository.findAll());

    }
```

O método `getLutadores()` está anotado com a `@GetMapping` do Spring, o que o configura como método de um *endpoint* de URI `/lutadores` cujo verbo HTTP é `GET`.

Esse método usa a `repository`, que é uma instância de `LutadorRepository`, para tentar recuperar todos os lutadores do banco

de dados. Caso encontre lutadores, devolve uma resposta com status *200 Ok* e com um JSON que representa uma lista de objetos do tipo `Lutador`.

Caso não seja encontrado o melhor lutador (caso de base de dados vazia), o método fará o *endpoint* retornar um status *204 No Content* e corpo vazio.

Endpoint *POST /lutadores {JSON}* da `LutadorController`

```
public class LutadorController {

    // atributos
    // método getLutadores()

    @PostMapping
    public ResponseEntity criarLutador(@RequestBody Lutador lutador) {
        return
ResponseEntity.status(HttpStatus.CREATED).body(this.service.save(lutador))
;
    }
```

O método `criarLutador()` está anotado com a `@PostMapping` do Spring, o que o configura como método de um *endpoint* de URI `/lutadores` cujo verbo HTTP é `POST`. É necessário enviar um corpo na requisição: um JSON que represente uma instância de `Lutador`.

Esse método usa o `service`, que é uma instância de `LutadorService`, para tentar inserir o lutador no banco de dados. Caso a operação ocorra sem erros, devolve uma resposta com status *201 Created* e com um JSON que representa a instância do `Lutador` recém-salvo no banco.

Caso ocorra algum erro na operação, o *endpoint* retornará o status e mensagem de erro definidos no método `save(Lutador lutador)` da `LutadorService`, já explicado neste capítulo.

Arquivo `application.properties` do projeto Spring Boot

Além das classes, o projeto também possui um `application.properties` que, por padrão, fica em `/src/main/resources`. É o nome padrão que o arquivo de definição de *properties* (propriedades) de um projeto que usa Spring. Ele será usado para configurar o acesso ao banco de dados e uma mensagem que deverá ser obtida na `LutadorService` também no teste de integração. Seu conteúdo está a seguir:

```
maximo.lutas=50
erro.maximo-lutas=O máximo de lutas ultrapassado
```

Testes de integração Spring Boot para a Service

O primeiro teste de integração será com a `LutadorServiceTest`, que deve ficar no diretório padrão de testes, o `src/test/groovy`. Nela, vamos testar os seguintes cenários:

- O método `save()` deve lançar exceção com status 400 se a quantidade de lutas for maior que o máximo configurado;
- O método `save()` deve retornar o `Lutador` salvo se não houver erro.

Classe `LutadorServiceTest` (teste de integração com *Spring Module*)

Vamos ver como seria a estrutura inicial da classe de testes `LutadorServiceTest`:

```
// pacote e imports Spring e JPA
@SpringBootTest(classes = LutadorService)
class LutadorServiceTest extends Specification {

    @Autowired
    LutadorService service

    @Value('${maximo.lutas}')
    private int maximoLutas

    @SpringBean
```

```
    LutadorRepository repository = Mock()

    // métodos de teste
}
```

Note que classe deve estender a `Specification` do Spock, como os demais testes que já estudamos até aqui.

A anotação `@SpringBootTest` indica que o teste será executado com auxílio do framework de testes do Spring. Seu atributo `classes` indica quais classes devem ser usadas pelo Spring durante o teste. No caso, indicamos apenas a `LutadorService`.

Todas as classes que indicadas no `classes` do `@SpringBootTest` podem ser injetadas via `@Autowired` na classe de testes. Foi assim que injetamos a `LutadorService service` na classe. Ou seja, o objeto `service` será uma instância real, um componente (*bean*) Spring de verdade da `LutadorService`. NÃO será um dublê (*mock*).

O atributo `maximoLutas` terá seu valor injetado a partir da propriedade `maximo.lutas` devido ao uso da anotação `@Value`. Essa propriedade deve estar no arquivo `application.properties` ou em outro caso use *profiles* do Spring (detalhes no tópico ***Profiles e properties do projeto Spring nos testes de integração*** deste capítulo). Note que usamos a anotação `@Value` do Spring com seu valor entre aspas simples. **Esse detalhe é importante**: se usar aspas duplas, o Groovy vai tentar executar como expressão o que estiver entre `${}`, por isso deve-se usar aspas simples nessa anotação em testes com Spock.

A anotação `@SpringBean` é do Spock e permite a injeção de *mocks* na classe de teste. Como usamos sobre um atributo do tipo `LutadorRepository`, todos objetos Spring reais que porventura injetem a `LutadorRepository` na `LutadorServiceTest` injetarão esse *mock* dela e não o componente (*bean*) Spring real. Mesmo usando essa anotação, temos que iniciar o objeto com o valor `Mock()`, que é um método do Spock.

Teste para o cenário *"O método `save()` deve lançar exceção com status 400 se a quantidade de lutas for maior que o máximo configurado"*

Vamos ver como seria o método de teste para esse cenário:

```
// pacote e imports Spring e JPA
@SpringBootTest(classes = LutadorService)
class LutadorServiceTest extends Specification {

    // atributos

    def "save() deve lançar exceção com status 400 se a quantidade de lutas for maior que o máximo configurado"() {
        given:
        def lutador = new Lutador(vitorias: maximoLutas, derrotas: maximoLutas, nocontest: maximoLutas)

        when:
        service.save(lutador)

        then:
        def exception = thrown(ResponseStatusException)
        exception.rawStatusCode == 400
        exception.reason == service.mensagemErroMaximoLutas
    }
}
```

No bloco `given`, criamos um `Lutador` com vitórias, derrotas e *no contests* todos iguais à quantidade máxima de lutas por lutador. Isso certamente provocará uma falha da validação, afinal teremos 3x mais lutas que o máximo permitido.

No bloco `when`, invocamos o método `save()` da `LutadorService` passando como argumento o `Lutador` recém-criado.

No bloco `then`, as verificações foram:

- Deve ocorrer uma exceção do tipo `ResponseStatusException`;
- O status de resposta da exceção deve ser 400;

- A mensagem de erro da exceção deve ser a mesma definida no atributo `mensagemErroMaximoLutas` da `LutadorService`.

Teste para o cenário *"O método `save()` deve retornar o `Lutador` salvo se não houver erro"*

Vamos ver como seria o método de teste para esse cenário:

```
// pacote e imports Spring e JPA
@SpringBootTest(classes = LutadorService)
class LutadorServiceTest extends Specification {

    // atributos e método do cenário anterior

    def "save() deve retornar o Lutador salvo se não houver erro"() {
        given:
        def atributos = [vitorias: 0, nocontest: 0, derrotas: 0]
        def lutadorEnviado = new Lutador(atributos)
        def lutadorEsperado = new Lutador(atributos+=[id: 100])

        and:
        repository.save(lutadorEnviado) >> lutadorEsperado

        when:
        def lutadorResultado = service.save(lutadorEnviado)

        then:
        lutadorResultado.properties == lutadorEsperado.properties
    }
}
```

No bloco `given`, primeiro criamos 2 objetos do tipo `Lutador`, ambos sem lutas. O `lutadorEnviado` não tem `id` e o `lutadorEsperado` tem `id` igual a `100`.

Em seguida, configuramos o *mock* `repository` para que, ao ter seu método `save()` invocado com o `lutadorEnviado`, retorne o `lutadorEsperado`.

No bloco `when`, invocamos o método `save()` da `LutadorService` passando como argumento o `lutadorEnviado`. Guardamos o retorno do método em um novo objeto, o `lutadorResultado`.

No bloco `then`, verificamos se todos os atributos de `lutadorResultado` são iguais aos de `lutadorEsperado`. Assim, garantimos que a `LutadorService` não dê erro nem altere nada no lutador gerado pelo `save()` da `repository` caso o lutador seja válido.

Testes de integração Spring Boot para a Controller

O próximo teste de integração será com a `LutadorControllerTest`, que também deve ficar no diretório padrão de testes, o `src/test/groovy`. Nela, vamos testar os seguintes cenários:

- Deve retornar status 204 e sem corpo quando não existirem lutadores;
- Deve retornar 200 e trazer todos os lutadores e no formato correto quando existirem lutadores;
- Deve retornar status 400 e a mensagem esperada quando falha em criar `Lutador`;
- Deve retornar status 201 com JSON correto quando cria um `Lutador`.

Esses cenários podem ser testados de duas formas:

1. **Testando os *endpoints* da *controller*:** nesse caso, criamos testes de unidade sobre seus *endpoints*. Nós inicamos a API parcialmente e testamos seus *endpoints* fazendo chamadas REST e analizando suas respostas HTTP;

2. **Testando a *controller* como uma classe:** nesse caso, criamos testes de unidade e de integração sobre seus métodos como faríamos como qualquer outro tipo de classe.

Como testes com classe já foram feitos para a `LutadorService`, vamos usar a primeira abordagem (teste dos *endpoints*). A seguir, vamos analisar o código dessa classe:

Classe `LutadorControllerTest` (teste de integração com *Spring Module*)

```
// pacote e imports Spring e JPA
@WebMvcTest(LutadorController)
@Stepwise
class LutadorControllerTest extends Specification {

    @Autowired
    MockMvc mockMvc

    @SpringBean
    LutadorService mockService = Mock()

    @SpringBean
    LutadorRepository mockRepository = Mock()

    @Value('${maximo.lutas}')
    private int maximoLutas

    @Value('${erro.maximo-lutas}')
    private String mensagemErroMaximoLutas

    // métodos de teste
}
```

A classe `LutadorControllerTest` está anotada com `@WebMvcTest` do Spock, que indica que a classe deve iniciar um contexto do Spring Boot para a realização de testes de integração que permita o teste de *endpoints* de *controllers*. Dentro dela, indicamos qual *controller* queremos testar. Aqui indicamos a `LutadorController`.

A anotação `@Stepwise` (já explicada no capítulo 9, *Recursos avançados*) foi usada apenas porque a sequência da execução dos testes é necessária para o correto funcionamento dos testes de integração de nosso exemplo.

A `@WebMvcTest` permite que possamos injetar a `MockMvc` na classe de testes. Objetos desse tipo permitem que façamos chamadas REST

para *endpoints* de uma *controller*, assim como nos permite analisar a resposta das chamadas. Tal como a `@SpringBootTest` permite a injeção de propriedades por meio da anotação `@Value`.

Os atributos `LutadorService mockService` e `LutadorRepository mockRepository` são *mocks*, devido à anotação `@SpringBean`, como já explicado na classe `LutadorServiceTest`.

Os atributos `maximoLutas` e `mensagemErroMaximoLutas` terão seus valores injetados conforme as propriedades `maximo.lutas` e `erro.maximo-lutas`, respectivamente, devido à anotação `@Value`, como já explicado na classe `LutadorServiceTest`.

Teste para o cenário *"Deve retornar status 204 e sem corpo quando não existirem lutadores"*

```
// pacote e imports Spring e JPA

import static
org.springframework.test.web.servlet.request.MockMvcRequestBuilders.*;
import static
org.springframework.test.web.servlet.result.MockMvcResultMatchers.*;

class LutadorControllerTest extends Specification {

    // atributos

    def 'Deve retornar status 204 e sem corpo quando não existirem
lutadores'() {
        when:
        def execucao = mockMvc.perform(get("/lutadores"))

        then:
        execucao
            .andExpect(status().isNoContent()) // ou
.andExpect(status().is(204))
```

```
                .andExpect(content().string(""))
    }
}
```

No bloco `when`, usamos o `mockMvc` para fazer uma chamada `GET` para `/lutadores`. O método `get()` foi importado de forma estática de `MockMvcRequestBuilders`. Guardamos o resultado da execução no objeto `execucao`.

No bloco `then`, fazemos as seguintes verificações:

- O status de resposta da chamada foi 204? O método `status()` foi importado de forma estática de `MockMvcResultMatchers`;
- O corpo da resposta está vazio? O método `content()` foi importado de forma estática de `MockMvcResultMatchers`.

Mas por que a chamada de endpoint desse método vai retornar 204 e sem corpo? Porque esse *endpoint* é implementado pelo método `getLutadores()` da `LutadorController`. Este, para definir se retorna status 200 ou 204, com ou sem corpo de resposta, usa os métodos `count()` e `findAll()` da `LutadorRepository`.

Ora, se a `LutadorRepository` é um *mock* durante a execução deste teste, logo ela não vai de fato para o banco de dados. O método `count()` retorna um `long` e o `findAll()` retorna uma `List`. Como já explicado no capítulo sobre *mocks*, seus retornos padrão serão, respectivamente, `0` (zero) e `null`. E como não definimos o que esses métodos do *mock* farão nesse teste, logo serão usados seus retornos padrão. Assim, como dentro do `getLutadores()` da `LutadorController` a `repository` vai retornar `0` em seu `count()`, o *endpoint* deveria retornar status 204 e sem corpo.

Teste para o cenário *"Deve retornar 200 e os lutadores com os dados do banco quando existirem Lutadores"*

```
// pacote e imports Spring e JPA
class LutadorControllerTest extends Specification {

    // atributos e método de teste do cenário anterior
```

```
    def 'Deve retornar 200 e os lutadores com os dados do banco quando existirem Lutadores'() {
        given:
        def lutadores = [
            new Lutador(id: 1, nome: 'lutador Ama', peso:80.0),
            new Lutador(id: 2, nome: 'lutador Flex', peso:110.0)
        ]
        def lutadoresJSON = JsonOutput.toJson(lutadores)

        and:
        mockRepository.count() >> lutadores.size()
        mockRepository.findAll() >> lutadores

        when:
        def execucao = mockMvc.perform(get("/lutadores"))

        then:
        execucao
                .andExpect(status().isOk()) // ou .andExpect(status().is(200))
                .andExpect(content().string(lutadoresJSON))
    }
}
```

No bloco `given`, definimos a lista `lutadores` que será no "banco de dados de mentirinha", com 2 lutadores. A variável `lutadoresJSON` será a lista convertida para um texto no formato JSON. Usaremos esse JSON na etapa de verificações do teste.

Também definimos que o método `count()` da `mockRepository` deve retornar o tamanho da `lutadores` e que o método `findAll()` da `mockRepository` deve retornar a lista `lutadores`. Ou seja, nosso `mockRepository` vai se comportar como se estivesse indo a um banco de dados e que nele temos 2 registros de lutadores.

No bloco `when`, usamos o `mockMvc` para fazer uma chamada **`GET`** para **`/lutadores`**. Guardamos o resultado da execução no objeto `execucao`.

No bloco `then`, fazemos as seguintes verificações:

- O status de resposta da chamada foi 200?
- O corpo da resposta contém o JSON esperado (ou seja, o mesmo valor de `lutadoresJSON`)? Aqui queremos garantir que a *controller* não altera o conteúdo entregue pelo `findAll()` da `LutadorRepository`.

Teste para o cenário *"Deve retornar status 400 e a mensagem esperada quando falha em criar `Lutador`"*

```
// pacote e imports Spring e JPA
class LutadorControllerTest extends Specification {

    // atributos

    def 'Deve retornar status 400 e a mensagem esperada quando falha em criar Lutador'() {
        given:
        def lutadorJSON = JsonOutput.toJson(new Lutador(vitorias: maximoLutas+1))

        and:
        mockService.save(_) >> {
            throw new ResponseStatusException(HttpStatus.BAD_REQUEST, mensagemErroMaximoLutas)
        }

        when:
        def execucao = mockMvc.perform(post("/lutadores")
                .contentType(MediaType.APPLICATION_JSON)
                .characterEncoding(StandardCharsets.UTF_8)
                .content(lutadorJSON))

        then:
        execucao
                .andExpect(status().isBadRequest()) // ou .andExpect(status().is(400))
                .andExpect(status().reason(mensagemErroMaximoLutas))
```

```
    }

    // demais métodos de teste
}
```

No bloco `given`, criamos a variável `lutadorJSON`, que é um texto em JSON criado a partir de um `Lutador` com mais lutas do que o máximo permitido.

Também definimos que o método `save()` da `mockRepository` deve lançar uma `ResponseStatusException` com status 400 e o valor do atributo `mensagemErroMaximoLutas` como mensagem de erro.

No bloco `when`, usamos o `mockMvc` para fazer uma chamada **`POST`** para **`/lutadores`** com `content-type` do tipo JSON e mapa de caracteres `UTF-8`. O corpo da requisição é o JSON de `lutadorJSON`. Guardamos o resultado da execução no objeto `execucao`. O método `post()` foi importado de forma estática de `MockMvcRequestBuilders`.

No bloco `then`, fazemos as seguintes verificações:

- O status de resposta da chamada foi 400?
- O mensagem de erro do corpo da resposta é igual ao valor de `mensagemErroMaximoLutas`, ou seja, da propriedade `erro.maximo-lutas`?

Teste para o cenário *"Deve retornar status 201 com JSON correto quando cria um `Lutador`"*

```
// pacote e imports Spring e JPA
class LutadorControllerTest extends Specification {

    // atributos

    def 'Deve retornar status 201 com JSON correto quando cria um
Lutador'() {
        given:
        def atributos = [nome: 'Ze Ruela', peso: 80, altura: 1.9]
```

```
        def lutadorRequisicao = new Lutador(atributos)
        def lutadorResposta = new Lutador(atributos += [id:100])

        and:
        def lutadorRequisicaoJSON = JsonOutput.toJson(lutadorRequisicao)
        def lutadorEsperadoJSON = JsonOutput.toJson(lutadorResposta)

        and:
        mockService.save(_) >> lutadorResposta

        when:
        def execucao = mockMvc.perform(post("/lutadores")
                .contentType(MediaType.APPLICATION_JSON)
                .characterEncoding(StandardCharsets.UTF_8)
                .content(lutadorRequisicaoJSON))

        then:
        execucao
            .andExpect(status().isCreated()) // ou
.andExpect(status().is(201))
            .andExpect(content().string(lutadorRespostaJSON))
    }

    // demais métodos de teste
}
```

No bloco `given`, criamos 2 objetos do tipo `Lutador`, ambos com os mesmos peso e altura. O `lutadorRequisicao` não tem `id` e o `lutadorResposta` tem `id` igual a `100`. Logo em sequida, criamos os textos no formato JSON `lutadorRequisicaoJSON` e `lutadorEsperadoJSON` criados a partir de `lutadorRequisicao` e `lutadorResposta`, respectivamente. Também definimos que o método `save()` da `mockRepository` deve retornar o `lutadorResposta` quando invocado com qualquer valor de argumento.

No bloco `when`, usamos o `mockMvc` para fazer uma chamada **POST** para **/lutadores** com `content-type` do tipo JSON e mapa de caracteres `UTF-8`. O corpo da requisição é o JSON de

`lutadorRequisicaoJSON`. Guardamos o resultado da execução no objeto `execucao`.

No bloco `then`, fazemos as seguintes verificações:

- O status de resposta da chamada foi 201?
- O corpo da resposta é igual ao JSON de `lutadorEsperadoJSON`?

12.3 Testes de integração em um projeto Spring MVC

O Spring MVC é um framework muito usado para criar projetos web com a arquitetura **monolítica**, ou seja, projetos que possuem desde a camada de acesso ao banco de dados até os artefatos de front-end. Assim como o Boot, o MVC possui uma série de componentes que abstraem boa parte do trabalho repetitivo e perigoso que envolve criar esse tipo de sistema.

Os cenários testados de Spring MVC serão:

- Deve retornar status 200 e redirecionar para JSP correto;
- Deve retornar status 400 quando o parâmetro " `usuario` " não for enviado.

Classe MVC Controller do projeto Spring

Neste estudo de caso, vamos apenas testar um *controller* do Spring MVC. Os demais componentes (`Service`, `Repository`, `Value` etc.) poderiam ser testados exatamente como fizemos no projeto com Spring Boot.

A classe que será usada como exemplo é a `LutadorMvcController`, cujo código está a seguir:

```
// pacote e imports do Spring
@Controller
```

```
public class LutadorMvcController {

    @GetMapping("/home")
    public ModelAndView home(@RequestParam String usuario)     {
        // ações com o argumento 'usuario'
        return new ModelAndView("home");
    }
}
```

A classe `LutadorMvcController` está anotada com `@Controller` do Spring, o que a transforma em um *controller* do Spring MVC. É um tipo de componente normalmente usado para gerar respostas HTTP com páginas HTML no corpo da resposta, mas que também pode gerar arquivos binários ou até textos simples.

O método `home()` está anotado com a `@GetMapping` do Spring, o que o configura como método de um *endpoint* de URI `/home` cujo verbo HTTP é `GET`. Esse método espera um parâmetro de requisição chamado `usuario` e retorna um `ModelAndView` cuja *view* será a página `src/main/webapp/WEB-INF/home.jsp`. O único argumento do método está anotado com a `@RequestParam` do Spring. Isso indica que, quem quiser chegar a esse método, terá que *obrigatoriamente* informar o parâmetro de requisição `usuario`.

Arquivo `application.properties` do projeto Spring MVC

Além da classe `Controller`, o projeto precisa de novas propriedades no `application.properties`. Elas serão usadas para configurar o diretório e extensão padrões para as *views* dos *controllers* do projeto. São elas:

```
spring.mvc.view.prefix=/WEB-INF/
spring.mvc.view.suffix=.jsp
```

O valor `/WEB-INF/` da chave `spring.mvc.view.prefix` é a partir de `src/main/webapp`. A chave `spring.mvc.view.suffix` indica a extensão padrão para os arquivos de *view* no projeto.

Arquivo `home.jsp` do projeto Spring MVC

O conteúdo do arquivo `src/main/webapp/WEB-INF/home.jsp` é até irrelevante para nosso estudo de caso. Porém, para deixar você mais seguro(a), segue um exemplo do que poderia estar em seu conteúdo:

```
<html>
<body>
    <h2>Bem vindo ao sistema de Lutadores!</h2>
    <div>Que belo sistema de lutadores ;)</div>
</body>
</html>
```

Cenários do teste de integração Spring MVC

A classe do teste de integração é a `LutadorMvcControllerTest`, que deve ficar no diretório padrão de testes, o `src/test/groovy`. Nela, vamos testar os seguintes cenários:

- Deveria retornar status 200 e redirecionar para JSP correto;
- Deveria retornar status 400 quando o parâmetro `usuario` não for enviado.

A seguir, vamos analisar o código dessa classe.

Classe `LutadorMvcControllerTest` (teste de integração com *Spring Module*)

```
// pacote e imports Spring
@WebMvcTest(LutadorMvcController)
class LutadorMvcControllerTest extends Specification {

    @Autowired
    MockMvc mvc

    @Autowired
    LutadorMvcController controller
```

```
    @Value('${spring.mvc.view.prefix}')
    String viewPrefix

    // métodos de teste
}
```

A classe `LutadorMvcControllerTest` está anotada com `@WebMvcTest` do Spock. Essa anotação indica que a classe deve iniciar um contexto do Spring MVC para a realização de testes de integração entre os componentes da aplicação. Ou seja, **é essa anotação que permite testes de integração Spring MVC** em uma classe de testes Spock.

A `@WebMvcTest` deve ter uma classe marcada como *controller Spring MVC* dentro dela, ou seja, uma classe anotada com `@Controller`. Por isso usamos a `LutadorMvcController`.

O atributo `MockMvc mvc` é **ator-chave nos testes de integração para Spring MVC**. É com ele que testamos o comportamento dos métodos dos *controllers* de forma integrada. Por isso, ele deve estar anotado com `@Autowired`, para que forneça acesso a todos os componentes Spring do projeto durante a execução dos testes.

O atributo `LutadorMvcController controller` é simplesmente o *controller* que queremos testar. O atributo `String viewPrefix`, anotado com `@Value` do Spring, será usado para verificar se a *view* do retorno do método a ser testado no *controller* está configurada corretamente.

Teste para o cenário cenário *"Deve retornar status 200 e redirecionar para JSP correto"*

```
// pacote e imports Spring
class LutadorMvcControllerTest extends Specification {

    // atributos

    def 'Deve retornar status 200 e redirecionar p/ JSP correto'() {
        given:
```

```
        def destino = "home"

        when:
        def resposta = mvc.perform(get("/home").param("usuario", "Lady
Gaga"))

        then:
        resposta
            .andExpect(status().isOk()) // ou .andExpect(status().is(200))
            .andExpect(forwardedUrl("${viewPrefix}${destino}.jsp"))
    }
}
```

No bloco `given`, definimos qual o `destino` esperado.

No bloco `when`, solicitamos a realização da chamada ao método `home()` de forma indireta, indicando o verbo HTTP, a URI e o parâmetro de requisição (parâmetro `usuario` com valor `Lady Gaga`).

No bloco `then`, verificamos:

- Se o status da resposta é *200 OK*;
- Se a *view* corresponde a `/WEB-INF/` + `home` + `.jsp`. O método `forwardedUrl()` foi importado de forma estática de `MockMvcResultMatchers`.

Teste para o cenário *"Deve retornar status 400 quando o parâmetro `usuario` não for enviado"*

```
// pacote e imports Spring
class LutadorMvcControllerTest extends Specification {

    // atributos

    def 'deve retornar status 400 quando o parâmetro "usuario" não for
enviado'() {
        when:
        def resposta = mvc.perform(get("/home"))

        then:
```

```
        resposta.andExpect(status().isBadRequest()) // ou
resposta.andExpect(status().is(400))
    }
}
```

No bloco `when`, solicitamos a realização da chamada ao método `home()` de forma indireta, indicando o verbo HTTP, a URI e a ausência de parâmetros de requisição.

No bloco `then`, verificamos se o status da resposta é *400 Bad Request*. Isso porque o parâmetro de requisição está como obrigatório em `LutadorMvcController`.

12.4 Ciclo de vida do contexto Spring nos testes de integração

Quando criamos uma classe de testes de integração usando *Spring TestContext framework* e *Spring Module do Spock*, é importante ter ciência de que **cada classe de testes possui seu contexto Spring próprio**. Isso significa que:

- Os **componentes são compartilhados entre os métodos de teste de uma mesma classe**, de acordo com o escopo de cada componente. Por exemplo: *Services* são *Singletons*, portanto terão a mesma instância compartilhada entre todos os métodos de testes de uma mesma classe. No caso de componentes de escopo *request*, terão uma instância diferente por método de uma mesma classe;
- Os **componentes não são compartilhados entre classes** de testes de integração Spring. É possível ver a mensagem de

inicialização do Spring Boot no log do Maven sempre que uma classe de testes de integração é iniciada.

Esse comportamento se aplica à classes de testes anotadas com `@SpringBootTest` ou `@WebMvcTest` e está ilustrado na figura a seguir:

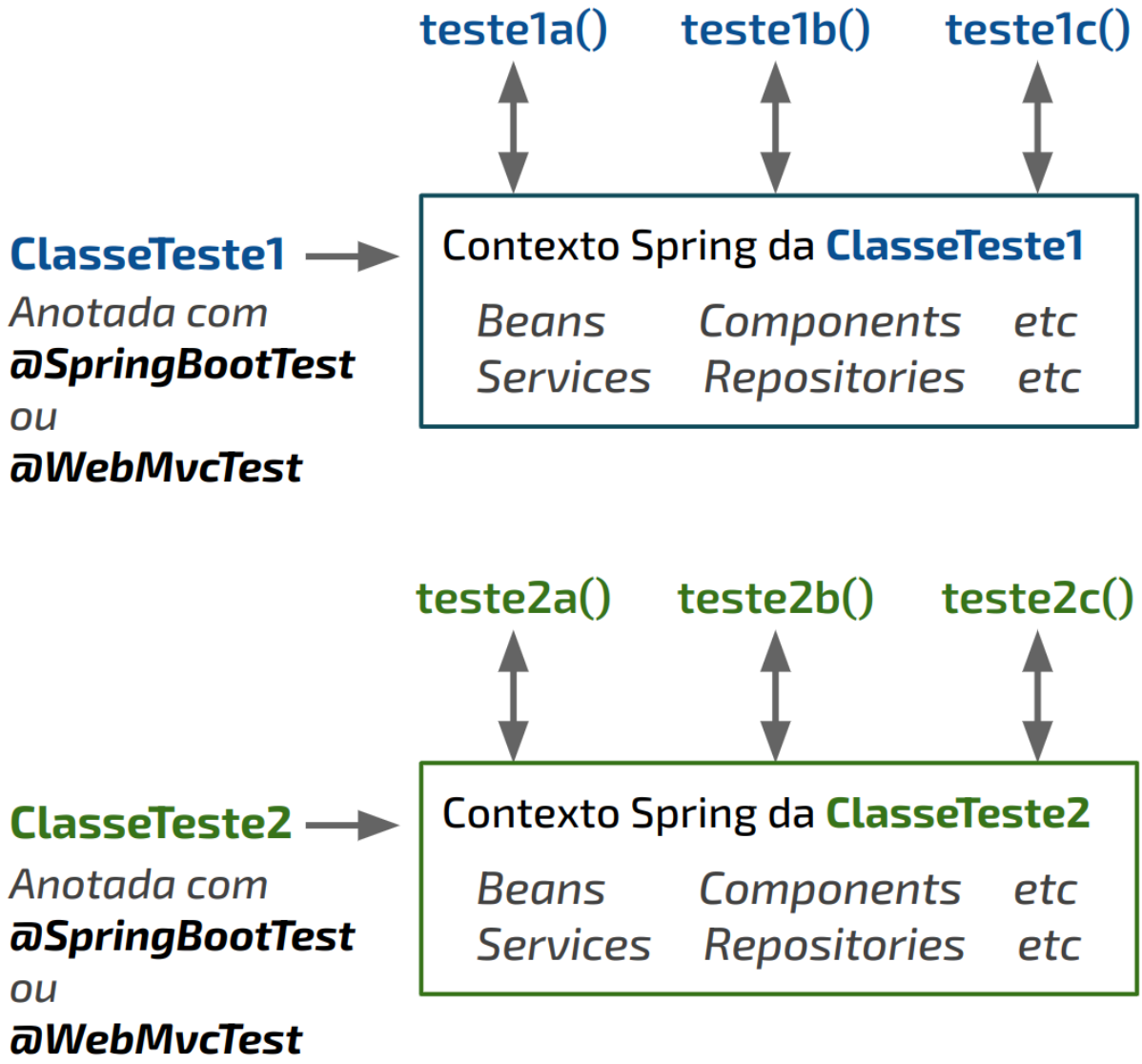


Figura 12.3: Ciclo de vida de contextos Spring em diferentes testes de integração

12.5 Profiles e properties do projeto Spring

nos testes de integração

Algumas das funcionalidades que mais dão flexibilidade de configuração em projetos com Spring são as *profiles* (perfis) e *properties* (propriedades), que podem ser configuradas de forma muito simples. A seguir, veremos como usar essas funcionalidades em testes de integração com Spock.

Definindo um arquivo de *properties* para os testes

É possível definir as *properties* (propriedades ou configurações) de um projeto que usa Spring escrevendo-as em um arquivo chamado `application.properties` (formato *Properties*, ou seja, *chave x valor*) ou `application.yml` (formato YAML, ou seja, combinação de listas, mapas e valores simples). Nesse arquivo, normalmente são definidas configurações como:

- Dados de acesso ao banco de dados;
- Dados de acesso a outros sistemas (por exemplo, servidor de e-mail);
- Textos de mensagens de validação.

Por padrão, o Spring procura um desses arquivos no diretório `/src/main/resources`. Porém, imagine que você precisa definir configurações apenas para os testes, tais como o uso de um banco de dados em memória, por exemplo. Para situações assim, é possível definir um arquivo de *properties* exclusivo para os testes de integração. Para isso, basta criar um `application.properties` (ou `.yml`) no diretório `/src/test/resources` para que ele seja automaticamente usado pelos testes de integração.

Atenção: caso exista um arquivo de configuração em `src/test`, os arquivos em `src/main` serão ignorados durante os testes de integração.

Escolhendo o *profile* para os testes

Um dos recursos mais interessantes do Spring é a facilidade em ter diferentes *profiles* (perfis) em um mesmo projeto. Isso permite, por exemplo, que a aplicação acesse diferentes bancos de dados, de acordo com o *profile* selecionado. Para que um projeto Spring tenha vários perfis configurados, basta que possua vários arquivos de *properties* com nome terminando em `-nome_do_perfil`.

Por exemplo, nosso projeto teria vários *profiles* caso ele tivesse os seguintes arquivos em `src/main/resources` (ou em `src/test/resources`):

- application.properties (*profile* `padrão`)
- application**-test**.properties (*profile* `test`)
- application**-dev**.properties (*profile* `dev`)
- application**-homol**.properties (*profile* `homol`)
- application**-prod**.properties (*profile* `prod`)

Quando criamos uma classe de testes de integração usando *Spring TestContext framework* e *Spring Module do Spock*, é possível indicar o *profile* que queremos usar na execução de uma classe de testes. Para isso, basta usar a anotação `@ActiveProfiles` sobre a definição da classe de testes. Poderíamos reconfigurar a `LutadorControllerTest` deste capítulo para tentar usar um *profile* chamado `test` durante a execução dos testes de integração. Então ela ficaria como no código a seguir:

```
@ActiveProfiles("test")
// demais anotações da classe
class LutadorControllerTest extends Specification {
    ...
}
```

Com a inclusão da anotação `@ActiveProfiles("test")` sobre a classe, todos os testes nela serão executados considerando as configurações no arquivo `application-test.properties` (ou `application-test.yml`) em `src/test/resources` **ou** em `src/main/resources`.

Ordem de busca de *properties* conforme *profile*

Imagine que você possui várias *properties* e *profiles* configurados. Ter configurações espalhadas em tantos arquivos pode causar confusões durante os testes. Para que você tenha certeza de qual arquivo de configuração será usado na busca de suas configurações, basta saber qual a sequência de arquivos nos quais uma propriedade é buscada pelo Spring.

Para melhor demonstrar isso, imagine que você solicitou a busca pela *property* `msg.erro.sistema` e sua classe de testes está configurada para o *profile* `test`. O Spring tentaria localizar essa *property* nas seguintes sequências durante a execução do teste de integração:

Cenário 1: existe pelo menos 1 arquivo de *properties* em `src/test/resources`

1. Arquivo `application-test.properties` (ou `.yml`) em `src/test/resources`;
2. Arquivo `application.properties` (ou `.yml`) em `src/test/resources`.

Caso a *property* `msg.erro.sistema` não seja encontrada em nenhum desses dois arquivos, uma exceção específica do Spring será lançada.

Cenário 2: não existe nenhum arquivo de *properties* em `src/test/resources`

1. Arquivo `application-test.properties` (ou `.yml`) em `src/main/resources`;
2. Arquivo `application.properties` (ou `.yml`) em `src/main/resources`.

Caso a *property* `msg.erro.sistema` não seja encontrada em nenhum desses dois arquivos, uma exceção específica do Spring será lançada.

Conclusão

Neste capítulo, vimos como configurar projetos Spring MVC e Spring Boot para que seja possível criar testes de integração com Spock usando o módulo Spring. Também estudamos um pequeno exemplo de teste de integração em cada um deles para demonstrar como usar *mocks* e a injeção de componentes Spring.

CAPÍTULO 13
Apêndice A - Guia de Groovy para desenvolvedores Java

Groovy é a linguagem oficial do Spock framework, por isso é fundamental ter uma proficiência mínima com ela. Este apêndice é para aquelas pessoas que possuem pouca ou nenhuma experiência com essa linguagem. É um guia focado para quem desenvolve em Java, para que façam o *"de x para"* (em Java é daquele jeito e em Groovy é desse) e tenham maior facilidade na escrita dos testes com Spock.

13.1 Como funciona o Groovy

A linguagem Groovy foi criada em 2003 por James Strachan e hoje possui seu código aberto, sendo mantida pela Apache Software Foundation desde 2015. É uma das linguagens cujo compilador gera arquivos executáveis pela Máquina Virtual Java, a **JVM**. Ou seja, arquivos escritos nessa linguagem, quando compilados, geram arquivos `.class` executáveis por qualquer JVM de versão de Java compatível. Na prática, isso significa que:

- Se uma IDE possui suporte a Java e Groovy, as classes escritas nessas duas linguagens conseguem trabalhar entre si de forma transparente;
- Bibliotecas escritas em Groovy podem ser usadas por projetos Java e vice-versa.

Esse funcionamento é representado na figura a seguir:

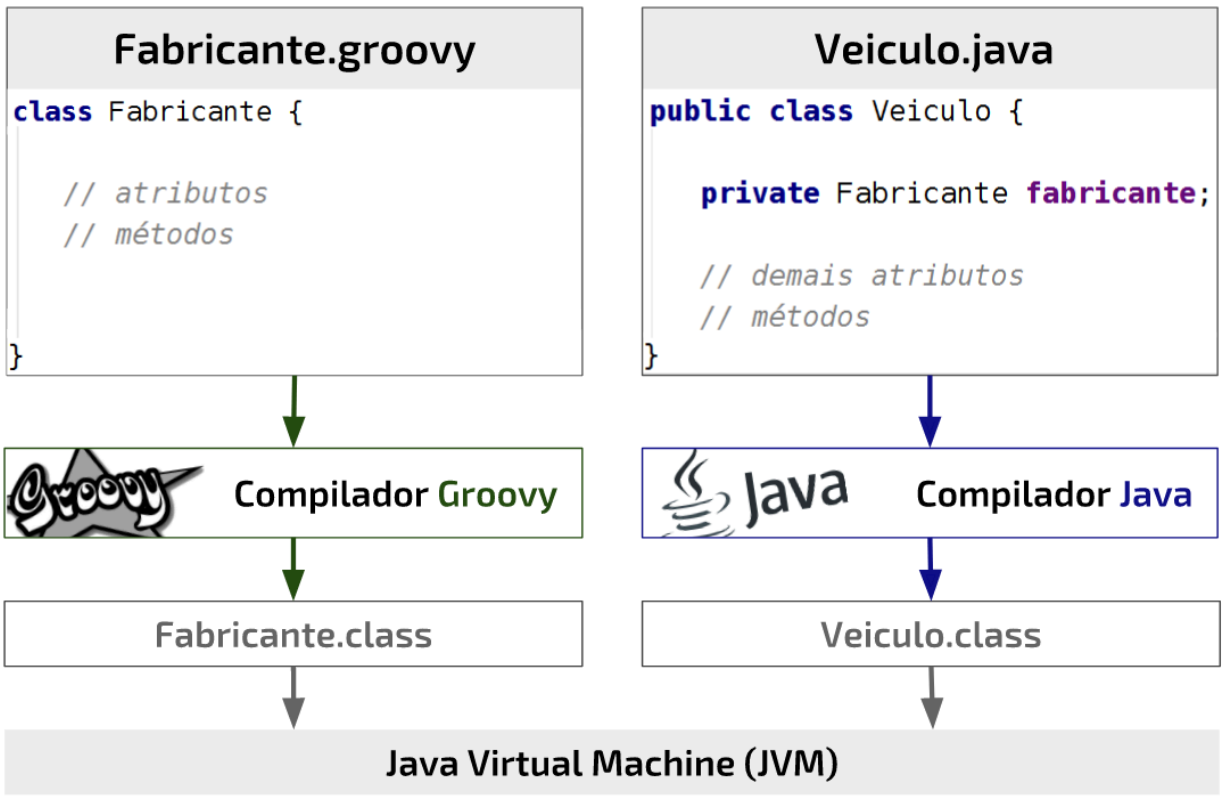


Figura 13.1: Funcionamento do Groovy na JVM

É por isso que o Spock, que possui como linguagem de programação o Groovy, pode ser usado para criar testes para projetos em Java, Scala, Kotlin, Jython, JRuby ou qualquer outra linguagem que compile para a JVM. Consequentemente, também é possível criar testes para projetos que usam qualquer framework Java, como Spring MVC/Boot, Play, Struts etc.

Uma característica que reduz a "curva de aprendizado" dessa linguagem para desenvolvedores Java é a proximidade entre as sintaxes e a possibilidade de escrever um código Groovy exatamente como escreveria em Java, que vai compilar e funcionar perfeitamente na imensa maioria dos casos. Ou seja, na dúvida em determinado trecho de código Groovy, faça exatamente como faria em Java. A seguir, alguns dos recursos e características do Groovy que podem ajudar muito na criação de testes com Spock.

13.2 Características e recursos do Groovy

Diferenças na sintaxe

A sintaxe do Groovy é muito parecida com a do Java. Porém, algumas diferenças na sintaxe do Groovy podem melhorar a produtividade no desenvolvimento. A seguir, várias dessas diferenças serão apresentadas.

Ponto e vírgula são opcionais no final da linha de código

Caso o desenvolvedor queira abrir mão de usar ponto e vírgula (**;**) ao final de cada linha, o compilador Groovy vai considerar a quebra de linha como o fim da linha de código. Porém, se ela for utilizada, será o final explícito da linha de código, como é em Java. É possível usar linhas com e sem ponto e vírgula no final em um mesmo arquivo Groovy sem problema algum.

`System.out` opcional

Caso seja necessário imprimir algo na saída padrão, é possível usar `System.out.print()` e `System.out.println()` como em Java. Porém, como o `System.out` é opcional em Groovy, podem ser substituídos apenas por `print()` e `println()`, respectivamente.

Parênteses são opcionais na invocação de métodos

Em Groovy, não é obrigatório o uso de parênteses na invocação de métodos, sejam eles com ou sem parâmetros. Como exemplo, temos o código a seguir, que imprime textos na saída padrão usando o `println` com e sem parênteses:

```
println("Não sou dono do mundo")
println "Mas sou filho do dono"
```

Outro exemplo: o código a seguir exemplifica a invocação de métodos da classe `java.lang.Math` sem parênteses, primeiro com um, depois com dois parâmetros:

```
double raiz144 = Math.sqrt 144 // retornará a raiz de 144: 12
double doisAoCubo = Math.pow 2,3 // retornará 2 ao cubo: 8
```

Ratificando: o não uso de parênteses é **opcional**. Caso você ache esse tipo de sintaxe confusa, pode usar os parênteses tal como usa na invocação de métodos em Java.

Por padrão, as classes são públicas

Quantas vezes você criou uma classe que não deveria ser `public`? Em Groovy, as classes são públicas por padrão. A classe `Futebolista` no exemplo a seguir será compilada como pública pela JVM:

```
// Futebolista.groovy
class Futebolista {  // em tempo de execução: "public class Futebolista"
  // atributos, métodos etc.
}
```

Por padrão, os atributos são privados

Quantas vezes você criou um atributo de instância que não deveria ser `private`? Em Groovy, os atributos são privados por padrão. Ou seja, na classe Groovy `Futebolista` a seguir, os atributos `nome` e `valorPasse` serão considerados privados pela JVM:

```
// Futebolista.groovy
class Futebolista {
  String nome // em tempo de execução: "private String nome"
  double valorPasse  // em tempo de execução: "private double valorPasse"
}
```

Por padrão, os métodos são públicos

Você criou muito mais métodos `public`, `private` ou `default` em suas classes? Em Groovy, os métodos são públicos por padrão. Ou seja, a classe Groovy `Futebolista` a seguir teria um método considerado público (`treinar()`) e outro privado (`fugirDaConcentracao()`) pela JVM:

```
// Futebolista.groovy
class Futebolista {
```

```
  String nome
  double valorPasse

  void treinar() {  // em tempo de execução: "public void treinar()"
    // treinando...
  }

  private void fugirDaConcentracao() {
    // fazendo a festa fora da concentração...
  }
}
```

É possível atribuir valores de atributos na criação de objetos

Em Java, quando queremos criar uma instância que já tenha determinados valores em atributos, é necessário criar construtores para isso. Em Groovy, toda classe possui um construtor que aceita um `Map` como argumento (porém, com colchetes opcionais). Assim, é possível indicar os argumentos que quiser e seus valores. Por exemplo, seria possível instanciar um objeto do tipo `Futebolista` como no código a seguir:

```
def jogadorA = new Futebolista(nome: 'Ze Boleiro', valorPasse: 100)
def jogadorB = new Futebolista(valorPasse: 100, nome: 'Ze Boleiro')
def jogadorC = new Futebolista(nome: 'Ze Boleiro')
def jogadorD = new Futebolista(valorPasse: 100)

def jogadorE = new Futebolista([nome: 'Ze Boleiro', valorPasse: 100])

Map mapaLoko = [nome: 'Ze Boleiro', valorPasse: 100] // Map criado em
algum lugar...
def jogadorF = new Futebolista(mapaLoko)
```

Métodos podem ter uma frase entre aspas como nome

Em classes de testes automatizados, normalmente um método testa um cenário. Imagine um cenário chamado "*o valor do passe do jogador não pode ser menor ou igual a 0 (zero)*". Se fôssemos

nomear um método da maneira convencional, ou seja, uma palavra só em ***camelCase***, seu nome seria algo como:

```
def valorPasseJogadorNaoAceitaZeroOuMenos() { ... }
```

Não acha que é um nome longo e difícil de ler? Uma alternativa seria reduzir o nome e pôr a descrição do cenário no `JavaDoc` do método. Porém, caso o teste falhe, é seu nome que é impresso no log da execução dos testes.

Usando Groovy, o nome do método poderia ser:

```
def 'valor do passe do jogador não pode ser menor ou igual a 0'() { ... }
```

Usando esse recurso, o cenário do teste fica literalmente descrito no nome do método. Outra vantagem é que, se o teste falhar, o log de execução conterá os cenários e não nomes de métodos longos e/ou difíceis de ler.

Podem ser usadas aspas simples ou duplas e até mesmo caracteres especiais (letras acentuadas, cê-cedilha etc.).

A tipagem pode ser estática ou dinâmica

Para reduzir o tempo de programação, em Groovy é possível criar objetos e métodos com o operador `def`, usando o recurso da **tipagem dinâmica**. Em caso de métodos, é possível omitir o tipo dos parâmetros. Vejamos exemplos a seguir.

Neste exemplo, foi usada a tipagem dinâmica para ambos os métodos. Em tempo de execução, a JVM vai considerar que eles retornam *Object*.

```
def metodoX(p1, p2) { // em tempo de execução: "public Object
metodoX(Object p1, Object p2)"
  // return ...;
}

def metodoY(def n1, def n2) { // em tempo de execução: "public Object
```

```
metodoY(Object n1, Object n2)"
  // return ...;
}
```

Se o método for `static`, até mesmo o `def` é opcional, como nos exemplos a seguir, cujos métodos serão considerados *públicos*, *estáticos* e com retorno *Object* em tempo de execução:

```
static metodoX(p1, p2) { // em tempo de execução: "public static Object
metodoX(Object p1, Object p2)"
  // return ...;
}

static def metodoY(def n1, def n2) { // em tempo de execução: "public
static Object metodoY(Object n1, Object n2)"
  // return ...;
}
```

Se você preferir, pode deixar a declaração do método ou dos argumentos estática, como nos exemplos a seguir:

```
Double metodoX(n1, n2) { // em tempo de execução: "public Double
metodoY(Object n1, Object n2)"
  // return ...;
}

Double metodoY(Double n1, n2) { // em tempo de execução: "public Double
metodoY(Double n1, Object n2)"
  // return ...;
}

void metodoZ(Double n1, Double n2) { // em tempo de execução: "public void
metodoZ(Double n1, Double n2)"
  // return ...;
}
```

Objetos também podem ser definidos com esse recurso, como no exemplo a seguir, onde primeiro é criado um `BigDecimal` declarado com tipagem dinâmica e, depois, um outro com tipagem estática:

```
def numero0 = new BigDecimal(0) // em tempo de execução: BigDecimal
numero0 = ...
BigDecimal numero1 = new BigDecimal(1)
```

Uma boa notícia para o desenvolvedor é que as principais IDEs Java do mercado – Eclipse, IntelliJ e NetBeans – sugerem os métodos e atributos do tipo usado na instanciação. Ou seja, no último código, após `numero0`, essas IDEs sugeririam métodos e atributos da classe `BigDecimal`.

Operador `==` faz o papel do `.equals()`

Em Java, o operador `==` verifica se os membros comparados são, literalmente, o mesmo objeto, a mesma instância. Para saber se possuem *conteúdo* igual, deve-se usar o método `.equals()`. Inclusive é comum que iniciantes em Java tentem comparar duas Strings de mesmo conteúdo com `==` e acabem recebendo um `false` como resultado.

Acreditando que a maioria dos desenvolvedores faz muito mais comparação de conteúdo do que de referência de instância em seus projetos, os arquitetos da linguagem Groovy definiram que, nela, o `==` correspondesse ao `.equals()` do Java. Vejamos o exemplo a seguir, onde `texto1` e `texto2` têm seus conteúdos comparados:

```
String texto1 = // valor recebido em tempo de execução
String texto2 = // valor recebido em tempo de execução
def teste = texto1 == texto2
// se o conteúdo de "texto1" e "texto2" forem iguais, "teste" será "true"
```

Caso exista a necessidade de saber se dois objetos são a mesma instância, deve-se usar o método `.is()`, conforme o exemplo a seguir:

```
String texto1 = // valor recebido em tempo de execução
String texto2 = // valor recebido em tempo de execução
String texto3 = texto1
def teste1e2 = texto1.is(texto2) // "false"
def teste1e3 = texto1.is(texto3) // "true"
```

Concatenação de valores String mais dinâmica e simples

Em Groovy, um objeto `String` pode ser instanciado com aspas simples, como no exemplo a seguir:

```
String poema = 'Nas curvas do teu corpo capotei meu coração'
```

Porém, ao se fazer isso, perde-se a possibilidade de usar a capacidade de **concatenações dinâmicas do Groovy**, que só é possível quando se usa aspas duplas. Basta usar o operador `$` e o nome do objeto a ser concatenado no texto. Vejamos o exemplo a seguir, onde `time1`, `time2` e `dias` serão concatenados no conteúdo de `anuncio`:

```
def time1 = 'Tabajara'
def time2 = 'Sayajins'
def dias = 7
def anuncio = "O jogo será entre $time1 e $time2. Faltam $dias dias, não percam!"
// "O jogo será entre Tabajara e Sayajins. Faltam 7 dias, não percam!"
```

A montagem de um texto pode incluir resultados da invocação de métodos e/ou de operações matemáticas. Assim como o recurso anterior, é possível **executar métodos e operações matemáticas e concatenar seus resultados** em String Groovy. Nesse caso, a expressão deve estar dentro de `${}`.

No exemplo a seguir, o resultado de `toUpperCase()` em `time1` e de `substring(4)` sobre `time2`, bem como o resultado de `valor` dividido por 2 serão concatenados no conteúdo de `anuncio`:

```
def time1 = 'Tabajara'
def time2 = 'Sayajins'
def valor = 20
def anuncio = "O jogo será entre ${time1.toUpperCase()} e ${time2.substring(4)}. Mulheres pagam ${valor/2}, não percam!"
// "O jogo será entre TABAJARA e jins. Mulheres pagam 10, não percam!"
```

Caso exista a necessidade de montar um texto que deva estar tão associado a um objeto ao ponto de seu conteúdo acompanhar alterações feitas nesse objeto, pode-se lançar mão de outro tipo de concatenação mais dinâmica ainda, **verificando o valor atual de um objeto sempre que a String for invocada**. Para isso, o objeto deve estar dentro de `${->}`.

No código do próximo exemplo, associamos `time1` e `time2` a `anuncio` de tal forma que, sempre que `anuncio` for usada, seu conteúdo pode variar, de acordo com o conteúdo de `time1` e `time2`:

```
def time1 = 'Tabajara'
def time2 = 'Sayajins'
def frase = "O jogo será entre ${-> time1} e ${-> time2}"
// nesse momento, "anuncio" é
// "O jogo será entre Tabajara e Sayajins"

time1 = 'Remo'
time2 = 'Paysandu'
// a partir desse momento, "anuncio" passa a ser
// "O jogo será entre Remo e Paysandu"
```

String podem ter múltiplas linhas sem concatenações

Quem já precisou escrever uma instrução SQL ou criar um *mock* de um JSON em Java já deve ter feito todo tipo de malabarismo para que uma String de várias linhas não fique tão ilegível no código. Em Groovy, existe o recurso de múltiplas linhas. Basta usar **três aspas simples ou duplas** para instanciar uma String.

No exemplo de String de múltiplas linhas a seguir, é criada uma instrução SQL com 3 linhas:

```
def consultaTimesSemTitulo = '''
select * from tb_time
where num_titulos = 0
order by dt_criacao desc
'''
```

Caso você precise criar uma String de múltiplas linhas com concatenação de métodos e/ou operações matemáticas, basta usar **três aspas duplas** nos limites da String. O exemplo a seguir monta uma instrução SQL de múltiplas linhas concatenando seu conteúdo com `uf`:

```
def uf = 'AM'
def consultaTimesAmazonas = """
select * from tb_time
where estado = ${uf}
"""
```

Esse recurso só existe no Java a partir da versão **15**.

Interoperabilidade com classes Java

A interoperabilidade de Groovy com classes Java é total, contanto que as versões de Java usadas na compilação de ambos sejam compatíveis. Trata-se da mesma restrição entre classes de arquivos escritos em Java.

Para exemplificar essa interoperabilidade, consideremos a classe Java `apploko.pjava.Futebolista`, que contém três atributos simples, um construtor que altera todos eles e seus *getters* e *setters*:

```
package apploko.pjava

// imports

public class Futebolista {
  private String nome;
  private double habilidade;
  private double velocidade;

  public Futebolista(String nome, double habilidade, double velocidade) {
    // atribuições simples
  }
```

```
  // getters e setters públicos de todos os atributos
}
```

A classe Groovy `FutebolistasLokos` a seguir usa a classe Java `apploko.pjava.Futebolista` de forma transparente: uma instância é criada usando seu construtor e um de seus métodos, o `setNome()`, é invocado.

```
import apploko.pjava.Futebolista
// demais imports

class FutebolistasLokos {
  Futebolista createHabilidoso() {
    def habilidoso = new Futebolista("Romário", 95, 90)
    habilidoso.setNome("Romário 11")
    return habilidoso
  }
}
```

Se o projeto utilizar Maven ou Gradle, qualquer classe das dependências também será acessível de forma transparente pelas classes Groovy.

`NullPointerException` é muito fácil de ser evitado

Se você já trabalha com Java há algum tempo, já deve ter visto o `java.lang.NullPointerException` (vulgo **NPE**) várias vezes quando executava seu projeto. Isso ocorre quando tentamos invocar um método de um objeto nulo (`null`), o que acontece com uma certa frequência porque é muito trabalhoso (e fácil de esquecer) verificar a cada *getXXX()* se não temos um `null`. Em Groovy, esse problema é simples de resolver, pois existe o chamado **operador de navegação segura** (*Safe navigation operator*), que previne o NPE apenas com o uso de `?` antes do `.`. Veja o próximo código:

```
  def jogador = // instanciado em tempo de execução a partir do banco de
dados, por exemplo
  def pais = jogador?.getTime()?.getPais()?.getNome();
```

Nesse código, caso `jogador` ou `getTime()` ou `getPais()` seja `null`, o objeto `pais` simplesmente receberá `null`, **sem NPE**. Caso o operador `?` não seja usado, o risco de NPE é o mesmo de uma classe Java.

Manipulação de coleções é muito simples

A manipulação de coleções em Java é considerada *verbosa* demais por pessoas desenvolvedoras acostumadas com linguagens como PHP, Python, Ruby e JavaScript. Em Groovy, a manipulação de coleções é bem simples, se comparada com Java, conforme apresentado a seguir.

Criando uma coleção com valores

Em Java, para criar uma coleção com valores deve-se recorrer a classes utilitárias, sejam elas do Java ou de bibliotecas de terceiros. Em Groovy, esse tipo de operação é muito simples, como nos exemplos a seguir.

No próximo código, há um exemplo da criação de uma `List` em Groovy. Ele cria uma instância de `ArrayList` com 4 (quatro) elementos:

```
def posicoes = ['goleiro', 'zagueiro', 'meia', 'atacante']
```

O interessante no código anterior é que as IDEs Eclipse e IntelliJ **inferem** o tipo de objeto da coleção quando se cria uma usando todos os elementos do mesmo tipo. Assim, quando tentar a recuperação de elementos dessa coleção, a IDE exibirá que o retorno seria do tipo `String`. Caso não consigam fazer a inferência, a coleção será tratada como de `Object`.

No próximo código, há um exemplo da criação de um `Map` em Groovy. Ele cria uma instância de `HashMap` com 3 (três) elementos, com as chaves `derrotas`, `empates` e `vitorias`, cujos valores são `0`, `1` e `3`, respectivamente:

```
def campanha = ['derrotas': 0, 'empates': 1, 'vitorias': 3]
```

Um detalhe bem útil em Groovy é a possibilidade de ser opcional o uso de aspas quando as chaves do mapa são *string*. Portanto, no código anterior todas as três chaves poderiam estar sem aspas, como no código a seguir.

```
def campanha = [derrotas: 0, empates: 1, vitorias: 3]
```

Criando coleções vazias

Em Java, para criar uma coleção vazia, devemos usar construtores como faríamos para qualquer outro tipo de classe. Em Groovy, é possível usar os mesmos construtores, porém as principais coleções podem ser criadas com o uso de operadores, como nos exemplos a seguir.

No código Groovy adiante, há um exemplo de criação de `List` **vazia**, implementada como `ArrayList`:

```
def posicoes = []
```

No código a seguir, há um exemplo de criação de `Map` **vazia** implementada como `HashMap` em Groovy:

```
def pontuacoes = [:]
```

Adicionando um elemento a uma coleção

Em Java, para adicionar elementos a uma coleção, devemos recorrer a métodos como `add()` ou `put()`. Em Groovy, é possível usar exatamente os mesmos métodos, porém também é possível usar **operadores**, como nos exemplos a seguir.

No próximo código, há um exemplo da criação de uma `List` vazia e posterior inclusão de elementos nela:

```
def posicoes = []
posicoes += 'goleiro'
```

```
posicoes += 'zagueiro'
```

Há três maneiras de incluir um elemento em um `Map` em Groovy, todas exemplificadas no código a seguir:

```
def campanha = [:]
campanha.derrotas = 0     // chave: 'derrotas' / valor: 0
campanha['empates'] = 1   // chave: 'empates' / valor: 1
campanha += [vitorias: 3] // chave: 'vitorias' / valor: 3
```

No último exemplo do código anterior, a instrução em si é um novo `Map`. Por exemplo, seria possível criar um novo mapa a partir de um existente apenas acrescentando um elemento, como no próximo código:

```
def campanhaOriginal = [vitorias:1, derrotas:0]
def novaCampanha = campanhaOriginal += [empates: 3]
// o valor de novaCampanha seria [vitorias:1, derrotas:0, empates:3]
```

Recuperação de itens de uma coleção

Em Java, para recuperar um item de uma coleção, devemos recorrer a métodos como o `get()`. Em Groovy, é possível usar exatamente os mesmos métodos, porém também é possível usar **operadores**, como nos exemplos a seguir.

No próximo código, há exemplos da recuperação de um elemento de uma `List`:

```
def posicoes = ['goleiro', 'zagueiro', 'meia', 'atacante']
println(posicoes[0]) // imprimiria 'goleiro'
println(posicoes[2]) // imprimiria 'meia'
```

Há duas maneiras de recuperar um elemento de um `Map` em Groovy, exemplificadas no código a seguir:

```
def campanha = [derrota: 0, empate: 1, vitoria: 3]
campanha.derrota    // 0
campanha['vitoria'] // 3
```

Recuperando o último elemento de uma lista ou vetor

Imagine que você precise do último elemento em uma lista, mas não sabe quantos elementos ela conterá no momento em que precisar dessa informação. Em Java, é preciso verificar o tamanho da lista e usar seu valor menos 1 para recuperar o último elemento de uma lista. Em Groovy, há um método chamado `last()`, que abstrai isso e recupera o último elemento de uma lista, como no exemplo a seguir:

```
def listaDinamica = // recuperada de forma dinâmica (ex: de um banco de dados)
def ultimo = listaDinamica.last()
```

O método `last()` **não existe em mapas**, estando disponível apenas para listas e vetores.

Técnicas de iteração simples e embarcadas para números, coleções e Strings

É uma tarefa muito comum realizar iterações (repetições) para resolver problemas computacionais. Essas iterações são comumente baseadas em valores numéricos, nos elementos de uma coleção ou nas linhas de um texto. A seguir, veremos como essas iterações mais comuns podem ser feitas em Groovy.

Iteração a partir de números

Na necessidade da repetição de um trecho de código determinada por um número **N** em Java, é necessário criar estruturas de repetição como `for`, `do-while` ou `while`. Em Groovy, os números inteiros possuem um recurso embarcado que atende facilmente a essa necessidade. É a *closure* `times{}`.

No exemplo a seguir, está programada uma repetição de um trecho de código por 11 vezes:

```
11.times{
    // o código aqui repetirá 11 vezes
```

```
    println(it) // o it começa do 0 e vai até 10
}
```

Dentro dela, é criada uma variável de contagem chamada `it` que começa sempre com valor 0 (zero) e vai até o valor numérico usado no `times{}` menos 1. No exemplo anterior, ela iria de 0 a 10, já que fizemos `11.times{}`.

É possível dar um nome personalizado para a váriavel de contagem caso não queira o nome `it`. Basta fazer como no código a seguir:

```
11.times{ numero ->
    // o código aqui repetirá 11 vezes
    println(numero) // o numero é começa do 0 e vai até 10
}
```

Em vez do número 11, poderíamos ter aplicado o mesmo recurso sobre uma variável do tipo `Integer`.

Caso seja necessário saber em que passo da iteração estamos, basta usar a variável `it`, que é **injetada automaticamente** na *closure* `times{}`. Ela vai **de 0 a N-1**, onde **N** é o número de iterações. O código do exemplo a seguir demonstra como poderíamos lançar mão desse recurso:

```
def jogadores = [jogador1, jogador2, jogador3]
jogadores.size().times{
    println(jogadores[it].getNome())
}
```

Por questões de legibilidade de código ou para o caso de iterações aninhadas, é possível definir um nome da variável do passo da iteração. No exemplo a seguir, usamos o nome `j`:

```
def jogadores = [jogador1, jogador2, jogador3]
jogadores.size().times{ j ->
    println(jogadores[j].getNome())
}
```

Iteração a partir de `List` e `Set`

Quando há repetição de um trecho de código para cada elemento de uma coleção em Java, é necessário criar estruturas de repetição como `for`, `do-while`, `while` ou usar os `streams` ou método `forEach()` do **Java 8**. Em Groovy, as coleções possuem um recurso embarcado que atende facilmente a essa necessidade. São as *closures* `each{}` e `eachWithIndex{}`, apresentadas e exemplificadas a seguir.

Neste exemplo, está programada uma repetição para os itens de uma `List`, mas que funcionaria da mesma forma para `Set`. O `it` dentro da *closure* é o nome que cada elemento da lista recebe na iteração:

```
def jogadores = [jogador1, jogador2, jogador3]
jogadores.each{
    println("Nome do Jogador: ${it.getNome()}")
}
```

Por questões de legibilidade de código ou para o caso de iterações aninhadas, é possível definir um nome do elemento na iteração. No exemplo a seguir, usamos o nome `jogador`:

```
def jogadores = [jogador1, jogador2, jogador3]
jogadores.each{ jogador ->
    println("Nome do Jogador: ${jogador.getNome()}")
}
```

Caso seja necessário saber em que passo da iteração estamos, podemos usar a *closure* `eachWithIndex{}`. Nesse caso, **é obrigatório** indicar os nomes do elemento e da variável que conterá o índice (passo) da iteração (que começa em 0), como demonstrado no código do exemplo a seguir, em que chamamos o elemento de `jogador` e o índice de `i`:

```
def jogadores = [jogador1, jogador2, jogador3]
jogadores.eachWithIndex{ jogador, i ->
    println("Nome do ${i+1} Jogador: ${jogador.getNome()}")
}
```

Iteração a partir de `Map`

Quando há repetição de um trecho de código para cada elemento de um mapa (`Map`) em Java, é necessário criar estruturas de repetição como `for` , `do-while` , `while` ou usar os `streams` ou método `forEach()` do Java 8. Em Groovy, as coleções possuem um recurso embarcado que atende facilmente a essa necessidade: é a *closure* `each{}` , apresentada e exemplificada a seguir.

Neste código, está programada uma repetição para os itens de um `Map` . O `it` dentro da *closure* é o nome que cada elemento do `Map` recebe na iteração. O atributo `key` é a **chave** e o `value` , o **valor** do elemento:

```
def campanha = [derrotas: 0, empates: 2, vitorias: 5]
println("Campanha:")
campanha.each{
    println("${it.key}: ${it.value}") // ex de saída: "vitorias: 5"
}
```

Por questões de legibilidade de código ou para o caso de iterações aninhadas, é possível definir um nome do elemento na iteração. No exemplo a seguir, usamos o nome `campanha` :

```
def campanha = [derrotas: 0, empates: 2, vitorias: 5]
println("Campanha:")
campanha.each{ campanha ->
    println("${campanha.key}: ${campanha.value}") // ex. de saída:
"vitorias: 5"
}
```

É possível ainda definir diretamente um nome para a chave e outro para o valor, como no próximo código, em que os chamamos de `resultado` e `quantidade` , respectivamente:

```
def campanha = [derrotas: 0, empates: 2, vitorias: 5]
println("Campanha:")
campanha.each{ resultado, quantidade ->
    println("${resultado}: ${quantidade}") // ex. de saída: "vitorias: 5"
}
```

Verificação de "verdadeiro/falso" expandida, porém facilitada

A verificação de **verdadeiro/falso** (*true/false*) é, provavelmente, a mais comum em testes automatizados. É possível fazer esse tipo de verificação de forma simplificada em Groovy conforme as situações a seguir.

`null` é `false` e não nulo é `true`

Qualquer objeto que seja `null`, quando testado em um `assert` ou `if` ou **operador ternário** ou atribuído a uma variável `boolean`, será interpretado como `false`. Veja o código Groovy de exemplo a seguir, cujo `assert` **falharia**:

```
def jogador = null
assert jogador
```

Nas mesmas situações recém-citadas, qualquer objeto **não nulo** é, **a princípio**, `true`. No código do exemplo a seguir, o `assert` **passaria**:

```
def jogador = new Futebolista()
assert jogador
```

Observe: foi dito que, **a princípio**, o resultado é `true`. Existem casos em que o objeto não é nulo, mas sua verificação booleana pode ser `false`. Essas exceções serão apresentadas a seguir.

`String` vazia é `false` e não vazia é `true`

Qualquer `String` que seja **vazia** (`""` ou `''`), quando testada em um `assert` ou `if` ou **operador ternário** ou atribuída a uma variável `boolean`, será interpretada como `false`. Veja o código Groovy de exemplo a seguir, cujo `assert` **falharia**:

```
def nomeNogador = ""
assert nomeJogador
```

Nas mesmas situações recém-citadas, qualquer `String` **não vazia** será considerada `true`. No código do exemplo a seguir, o `assert` **passaria**:

```
def nomeNogador = "Zé Loko"
assert nomeJogador
```

Número 0 (zero) é `false`, qualquer outro é `true`

Qualquer número, qualquer tipo numérico (inclusive `BigDecimal`) que seja **0** (zero), quando testado em um `assert` ou `if` ou **operador ternário** ou atribuído a uma variável `boolean`, será interpretado como `false`. Veja o código Groovy de exemplo a seguir, cujos `asserts` **falhariam**:

```
def numeroLoko1 = 0
def numeroLoko2 = 0.0
def numeroLoko3 = new BigDecimal(0)
assert numeroLoko1
assert numeroLoko2
assert numeroLoko3
```

Nas mesmas situações recém-citadas, qualquer número **diferente de zero, mesmo negativo**, será considerado `true`.

Coleções vazias são `false` e não vazias são `true`

Qualquer coleção (`List`, `Set` ou `Map`) que seja **vazia** (sem elementos), quando testada em um `assert` ou `if` ou **operador ternário** ou atribuído a uma variável `boolean`, será interpretada como `false`. Veja o código Groovy de exemplo a seguir, cujo `assert` **falharia**:

```
def jogadores = []
assert jogadores
```

Nas mesmas situações recém-citadas, qualquer coleção **não vazia** será considerada `true`. No código do exemplo a seguir, o `assert` **passaria**:

```
def campanha = [derrotas:0, vitorias:7]
assert campanha
```

Métodos embarcados para as conversões mais comuns

A conversão de tipos é uma operação muito comum nos problemas computacionais atuais, principalmente devido à grande quantidade de integração de sistemas que ocorre hoje. Em Java, as conversões são um tanto verbosas. Em Groovy há métodos de conversão embarcados para as conversões mais comuns do cotidiano, como exemplificado a seguir.

Conversão de `String` para qualquer tipo numérico

O tipo `String` em Groovy já possui métodos embarcados para a conversão para todos os tipos numéricos da plataforma Java. No código de exemplo a seguir, variáveis `String` são convertidas com sucesso para `Integer`, `Long`, `BigInteger`, `Float`, `Double` e `BigDecimal`:

```
def txtNumericoInteiro = '8'
Integer inteiro = txtNumericoInteiro.toInteger()
Long longo = txtNumericoInteiro.toLong()
BigInteger bigInteger = txtNumericoInteiro.toBigInteger()

def txtNumericoReal = '8.5'
Float flutuante = txtNumericoReal.toFloat()
Double duplo = txtNumericoReal.toDouble()
BigDecimal bigDecimal = txtNumericoReal.toBigDecimal()
```

Vale destacar que, caso o valor da `String` não contenha um número válido para a conversão solicitada, uma exceção será lançada.

Conversão de `String` para `Boolean`

O tipo `String` em Groovy já possui métodos embarcados para a conversão para `Boolean`. Os valores `"true"`, `"y"` e `"1"` são convertidos para `true`. Qualquer outro valor é convertido para `false`.

No código do exemplo de conversão a seguir, todas as conversões resultariam em `true`, portanto todos os `assert` **passariam**:

```
def textoBoleano = 'true'
assert textoBoleano.toBoolean()
textoBoleano = 'y'
assert textoBoleano.toBoolean()
textoBoleano = '1'
assert textoBoleano.toBoolean()
```

No código do exemplo de conversão a seguir, todas as conversões resultariam em `false`, portanto, todos os `assert` **falhariam**:

```
def textoBoleano = 'false'
assert textoBoleano.toBoolean()
textoBoleano = 'oi'
assert textoBoleano.toBoolean()
textoBoleano = ''
assert textoBoleano.toBoolean()
```

Acesso direto a métodos privados

Em Java, um método privado (`private`) não pode ser invocado, a não ser com uso da `Reflection API`, o que não é tarefa simples. Em Groovy, o acesso a métodos privados é transparente, sendo possível invocá-los como se fossem públicos.

O código do exemplo a seguir é de uma classe Java que possui o método privado `atualizarClassificacao()`, que é invocado por seus outros dois métodos que são públicos, o `gol()` e o `finalizar()`:

```
public class Partida {
    public void gol(Jogador autor) {
       // ações a cada gol...
```

```
        this.atualizarClassificacao();
    }

    public void finalizar() {
        // ações ao finalizar partida
        this.atualizarClassificacao();
    }

    private void atualizarClassificacao() {
        // ações da atualização da classificação
    }
}
```

Para invocar o método `atualizarClassificacao()` de uma instância de `Partida` a partir de uma classe Groovy, basta invocá-lo diretamente, como se fosse público.

O trecho de código Groovy a seguir instancia uma `Partida` e invoca seu método privado. As IDEs IntelliJ, Eclipse e NetBeans sugerem e aceitam a invocação de métodos privados durante a edição de código Groovy.

```
def partidaLoka = new Partida();
partidaLoka.atualizarClassificacao()
```

Esse recurso é muito útil em casos como o do último exemplo, em que métodos privados possuem muita relevância, pois permite a criação de vários cenários de teste diretamente para eles.

Alteração do comportamento de métodos em classes e objetos em tempo de execução

Em algumas situações, métodos podem fazer ações muito complexas e/ou que dependem de elementos externos. Como exemplos, temos: acesso a bancos de dados, consumo de APIs e uso de bibliotecas de terceiros.

Para casos como esses, pode-se **sobrescrever** o comportamento do método em tempo de execução em Groovy sem a necessidade de uma subclasse anônima nem de uma instância *mock*.

Tomemos como base a classe a seguir como exemplo, a `ClienteRestTimesFutebol` e seu método `getToken()`. Esse método solicitaria um *token* de autenticação junto a uma API a partir de um `usuario` e uma `senha`.

```
public class ClienteRestTimesFutebol {
    public String getToken(String usuario, String senha) {
        // chamada à API para obter 'token'
    }
}
```

No código a seguir, definimos que **uma determinada instância** de `ClienteRestTimesFutebol`, a que chamamos de `clienteLoko`, passa a ter um comportamento **diferente** para o método `getToken()`, enquanto a instância `clienteNormal` continua com seu método original:

```
def clienteLoko = new ClienteRestTimesFutebol()
cliente1.metaClass.getToken = { String u, String s ->
    return "fake-token"
}

def clienteNormal = new ClienteRestTimesFutebol()

def token1 = clienteLoko.getToken(null, null) // apenas retorna "fake-
token"
def token2 = clienteNormal.getToken(null, null) // invocaria a versão
original do "getToken()"
```

Caso seja necessário que todos os objetos de uma classe tenham o comportamento de um método alterado, é possível fazer esse ajuste **na classe**, consequentemente, **em todas as suas instâncias**. Como no exemplo a seguir, que mudaria o `getToken()` para qualquer instância de `ClienteRestTimesFutebol`:

```
ClienteRestTimesFutebol.metaClass.getToken = { String u, String s ->
    return "fake-token"
}
def cliente1 = new ClienteRestTimesFutebol()
def cliente2 = new ClienteRestTimesFutebol()

def token1 = cliente1.getToken(null, null) // apenas retorna "fake-token"
def token2 = cliente2.getToken(null, null) // apenas retorna "fake-token"
```

Getters **e** ***setters*** **podem ser invocados apenas com o nome do atributo**

Os métodos `getters` e `setters` fazem parte do dia a dia de quem desenvolve na plataforma Java. Fazem parte do padrão **JavaBeans** e são usados pelos principais frameworks Java (https://docs.oracle.com/javase/tutorial/javabeans/writing/properties.html). Seu uso é tão comum que as principais IDEs Java possuem assistentes para a rápida criação desses métodos. Em Groovy, existe a possibilidade de invocar esses métodos apenas usando o nome do atributo.

No exemplo a seguir, o atributo privado `valorPasse` deveria ser acessível com um `get` e um `set` escritos de maneira correta:

```
public class Futebolista {

  private double valorPasse;

  public double getValorPasse() {
      return this.valorPasse;
  }

  public void setValorPasse(double valorPasse) {
      this.valorPasse = valorPasse;
  }
}
```

Para acessar os `setValorPasse()` e `getValorPasse()` em uma classe Groovy, poderíamos escrever um código como o do exemplo a

seguir:

```
Futebolista jogadorLoko = // obtendo o Futebolista
jogadorLoko.valorPasse = 50000000
def passeDoLoko = jogadorLoko.valorPasse
```

Apesar de parecer que houve um acesso direto ao atributo, **não houve**. O compilador Groovy verifica se existe um `getValorPasse()` na segunda linha e se existe um `setValorPasse(double parametro0)` na terceira. Assim, a versão compilada do último código, para a JVM, seria como o código a seguir:

```
Futebolista jogadorLoko = // obtendo o Futebolista
jogadorLoko.setValorPasse(50000000)
double passeDoLoko = jogadorLoko.getValorPasse()
```

Caso você ache melhor usar os *getters* e *setters* de forma explícita em suas classes Groovy, pode fazê-lo sem problema algum.

Conclusão

Os recursos apresentados neste capítulo são suficientes para a pessoa iniciante em Groovy, porém existem ainda muito mais recursos interessantes nessa linguagem. Se você desejar um guia completo, a documentação oficial da linguagem Groovy (http://groovy-lang.org/documentation.html) é muito bem escrita e rica em exemplos.

Capítulo 14
Apêndice B - Perguntas frequentes sobre o Spock

Seguem algumas dúvidas comuns que podem surgir no início dos estudos sobre o Spock.

1. Vou ter que jogar fora todos os testes em JUnit que tenho?

Não. Seus testes em JUnit 4+ continuam funcionando normalmente. Você apenas adicionou novas dependências ao projeto para usar Spock. É possível haver testes em Spock, JUnit 4 ou 5 (tentar usar JUnit 4 e 5 no mesmo projeto não funciona muito bem).

2. Ainda poderei usar Mockito, PowerMock, TestNG e afins?

Depende. Talvez sejam necessários ajustes nas dependências do Mockito, PowerMock, TestNG e afins para que continuem funcionando.

3. Terei que adotar Groovy no projeto, além dos testes?

Não. Não precisa adicionar nenhuma linha de código Groovy ao resto do seu projeto. Inclusive, as dependências que adicionou devido ao Spock podem todas estar apenas no escopo `test` do Maven/Gradle.

4. Posso usar o Spock para testar funcionalidades escritas em outras linguagens para a JVM como Kotlin, Scala, JRuby e Jython?

Sim. Se todas as dependências estiverem corretas e compatíveis e se seu ambiente de desenvolvimento estiver corretamente configurado, poderá testar classes escritas em outras linguagens da plataforma Java.

5. Posso escrever testes em projetos Android usando Spock?

Até versões mais antigas da plataforma de desenvolvimento Android, sim. Porém, como o plugin de uso do Groovy em projetos Android não é atualizado desde 2020, desde então não é possível criar testes com Spock em projetos Android.

6. Posso testar códigos escritos em linguagens fora da plataforma Java como Python, JavaScript e Go?

Não. Spock foi feito para testar artefatos da plataforma Java. Projetos de outras plataformas de desenvolvimento não podem ser testados com ele.

7. Meus projetos Java 5, 6 e 7 são compatíveis com Spock?

Sim, mas não com a versão usada nesta edição. Este livro tratou do Spock versão `2.3-groovy-4.0`, que é compatível com Java 8 ou superior. Desde a versão *1.2.x*, não existe mais compatibilidade com Java 6. Existem versões do Spock compatíveis com o Java 5, porém nessas versões mais antigas podem não existir várias das funcionalidades abordadas neste livro.

8. O Spock é compatível com Java 9+?

Até a versão `1.1.x`, não era. Porém, a partir da `1.2.x`, passou a ser compatível.

9. Posso usar o Spock nos meus projetos antigos que não usam Maven/Gradle?

Sim. Basta seguir o processo de instalação e controle de bibliotecas (arquivos `.jar`) no *classpath* que já usa no projeto. Atente para a compatibilidade com a versão do Java usada no projeto (veja a **pergunta 7**).

10. O JaCoCo e plugins de cobertura de testes das IDEs Eclipse, IntelliJ e NetBeans são compatíveis com Spock?

Sim. Os resultados do JaCoCo consideram os testes criados com Groovy/Spock, bem como os plugins das principais IDEs da plataforma Java.

11. Groovy tem fama de ser lenta. Spock é lento?

Mais lento que testes JUnit escritos em Java, sim. De fato, a compilação e a execução de classes Groovy são um pouco mais lentas que das escritas em Java. Porém, desde o Java 7 e com recursos do Groovy 2+ como o `CompileStatic` (http://docs.groovy-lang.org/latest/html/gapi/groovy/transform/CompileStatic.html), o desempenho de classes Groovy melhorou consideravelmente.

A diferença de desempenho não leva a outra ordem de grandeza. Exemplos:

- Testes em JUnit que levam poucos segundos para serem executados provavelmente levarão também poucos segundos em sua execução quando escritos em Spock;
- Suítes de testes em JUnit que levam pouco menos do que 1 minuto para serem executadas provavelmente levarão um pouco mais de 1 minuto em sua execução quando escritos em Spock.

12. Sou obrigado a usar o Groovy em produção caso adote o Spock?

Não. Se não usar classes Groovy no código de produção, mas somente no de testes, e tiver feito as configurações corretas no seu gerenciador de dependências, as bibliotecas do Groovy não serão empacotadas com o pacote de produção.

Capítulo 15
Referências

ANICHE, Maurício. *Testes automatizados de software*: Um guia prático. São Paulo: Casa do Código, 2015.

ASDF. *asdf*: The Multiple Runtime Version Manager. Disponível em: https://asdf-vm.com/. Acesso em: 3 set. 2023.

ATHAYDES, Renato. Spock Reports Extension. *GitHub*, c2023. Disponível em: https://github.com/renatoathaydes/spock-reports. Acesso em: 25 ago. 2023.

BECK, Kent. *Test-Driven Development*: By Example. Boston(EUA): Addison-Wesley, 2003.

CZARNECKI, Krzysztof; EISENECKER, Ulrich W. *Generative Programming*: Methods, Tools, and Applications. Boston(EUA): Addison-Wesley, 2000.

ECLIPSE, Foundation. 2023 Jakarta EE Developer Survey Report. *Eclipse Foundation*, 2023. Disponível em: https://outreach.eclipse.foundation/jakarta-ee-developer-survey-2023. Acesso em: 17 jan 2024.

FOWLER, Martin. Mocks Aren't Stubs. *martinFowler.com*, 2007. Disponível em: https://martinfowler.com/articles/mocksArentStubs.html. Acesso em: 16 ago. 2023.

GOOGLE Trends. *REST, SOAP - Comparison*. Disponível em: https://trends.google.com.br/trends/explore?date=2008-03-01%202023-09-04&q=%2Fm%2F03nsxd,%2Fm%2F077dn. Acesso em: 21 jul. 2019.

GUERRA, Eduardo. *Componentes Reutilizáveis em Java com Reflexão e Anotações*. São Paulo: Casa do Código, 2014.

GUTIERREZ, Felipe. *Pro Spring Boot*. New York (EUA): Apress, 2016.

KAPELONIS, Konstantinos. *Java Testing with Spock*. Shelter Island (EUA): Manning Publications, 2015.

KEITH, Mike; SCHINCARIOL, Merrick; KEITH, Jeremy;. *Pro JPA 2 - Mastering the JavaPersistence API*. New York (EUA): Apress, 2010.

MASON, Ross. How REST replaced SOAP on the Web: What it means to you. *InfoQ*, 2011. Disponível em: https://www.infoq.com/articles/rest-soap. Acesso em: 23 ago. 2023.

MESZAROS, Gerard. *Four Phase Test*. *xUnitPatterns.com*, 2011. Disponível em: http://xunitpatterns.com/Four%20Phase%20Test.html. Acesso em: 26 ago. 2023.

MESZAROS, Gerard. *Mocks, Fakes, Stubs and Dummies*. *xUnitPatterns.com*, 2011. Disponível em: http://xunitpatterns.com/Mocks,%20Fakes,%20Stubs%20and%20Dummies.html. Acesso em: 26 ago. 2023.

MESZAROS, Gerard. *XUnit Test Patterns*: Refactoring Test Code. New York (EUA): Pearson Education, 2007.

NORTH, Dan. Introducing BDD. *Dan North & Associates Limited*, 2006. Disponível em: https://dannorth.net/introducing-bdd/. Acesso em: 11 ago. 2023.

PEREIRA, Caio Ribeiro. *Construindo APIs REST com Node.js*. São Paulo: Casa do Código, 2016.

REMANI, Abdelmonaim. *The Art of Metaprogramming in Java*. 2012. Apresentação em slides. Disponível em: https://cdn.oreillystatic.com/en/assets/1/event/80/The%20Art%20of%20Metaprogramming%20in%20Java%20%20Presentation.pdf. Acesso em: 29 ago. 2018.

ROCHER, Graeme; LEDBROOK, Peter; PALMER, Marc; BROWN, Jeff; DALEY, Luke; BECKWITH, Burt. The Grails Framework 2.3.0. *Grails*, c2023. Disponível em: https://grails.github.io/grails2-doc/2.3.0/guide/introduction.html#whatsNew23. Acesso em: 2 set. 2023.

SAUDATE, Alexandre. *REST*: Construa APIs inteligentes de maneira simples. São Paulo: Casa do Código, 2013.

SDKMAN!. *SDKMAN! The Software Development Kit Manager*. Disponível em: http://sdkman.io/. Acesso em: 2 set. 2023.

SIÉCOLA, Paulo. *Web Services REST com ASP .NET Web API e Windows Azure*. São Paulo: Casa do Código, 2016.

SILVEIRA, Guilherme; AMARAL, Mário. *Java SE 8 Programmer I*: O guia para sua certificação Oracle Certified Associate. São Paulo: Casa do Código, 2015.

SPRING. Testing. *In*: *Spring Framework Documentation*. c2023. Disponível em: https://docs.spring.io/spring/docs/current/spring-framework-reference/testing.html. Acesso em: 1 set. 2023.

STATISTA. *Global market share held by the leading smartphone operating systems in sales to end users from 1st quarter 2009 to 2nd quarter 2018*. 2018. Disponível em: https://www.statista.com/statistics/266136/global-market-share-held-by-smartphone-operating-systems/. Acesso em: 3 set. 2023.

SUBRAMANIAM, Venkat. *Programming Groovy 2*: Dynamic Productivity for the Java Developer. Raleigh (EUA): Pragmatic Bookshelf, 2013.